UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Arquitetura Híbrida de Armazenamento no Contexto da Internet das Coisas

Bráulio Lívio Dias Cavalcante Junior

São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Bráulio Lívio Dias Cavalcante Junior

Arquitetura Híbrida de Armazenamento no Contexto da Internet das Coisas

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como requisito para a obtenção do título de mestre em Ciência da Computação.

Orientador: Prof. Dr. Douglas Dyllon Jeronimo de Macedo
Coorientador: Prof. Dr. Edward David Moreno Ordonez

São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE
PRÓ-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA
COORDENAÇÃO DE PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da Sessão Solene de Defesa da Dissertação do Curso de Mestrado em Ciência da Computação-UFS.
Candidato: BRAULIO LIVIO DIAS CAVALCANTE JUNIOR

Em vinte e nove dias do mês de maio do ano de dois mil e dezoito, com início às 14h00min, realizou-se na Sala de Seminários do DCOMP da Universidade Federal de Sergipe, na Cidade Universitária Prof. José Aloísio de Campos, a Sessão Pública de Defesa de Dissertação de Mestrado do candidato **Braulio Livio Dias Cavalcante Júnior**, que desenvolveu o trabalho intitulado: "***Arquitetura Híbrida de Armazenamento no Contexto da Internet das Coisas***", sob a orientação do Prof. Dr. Douglas Dyllon Jeronimo de Macedo. A Sessão foi presidida pelo Prof. Dr. Dyllon Jeronimo de Macedo (PROCC/UFS), que após a apresentação da dissertação passou a palavra aos outros membros da Banca Examinadora, Prof. Dr. Edward David Moreno Ordonez (PROCC/UFS), Prof. Dr. Ricardo José Paiva de Britto Salgueiro (PROCC/UFS) e, em seguida, ao Prof. Dr. Mário Antônio Ribeiro Dantas (UFJF). Após as discussões, a Banca Examinadora reuniu-se e considerou o mestrando) APROVADO "*(aprovado/reprovado*" SEM "*(com/sem)* " ressalvas. Atendidas as exigências da Instrução Normativa 01/2017/PROCC, do Regimento Interno do PROCC (Resolução 67/2014/CONEPE), e da Resolução nº 25/2014/CONEPE que regulamentam a Apresentação e Defesa de Dissertação, e nada mais havendo a tratar, a Banca Examinadora elaborou esta Ata que será assinada pelos seus membros e pelo mestrando.

Cidade Universitária "*Prof. José Aloísio de Campos*", 29 de Maio de 2018.

Prof. Dr. Dyllon Jeronimo de Macedo
(PROCC/UFS)
Presidente

Prof. Dr. Edward David Moreno Ordonez
(PROCC/UFS)
Examinador Interno

Prof. Dr. Ricardo José Paiva de Britto Salgueiro
(PROCC/UFS)
Examinador Interno

Prof. Dr.Mário Antônio Ribeiro Dantas (UFJF)
Examinador Externo

Braúlio Livio Dias Cavalcante Júnior
Candidato

Resumo

A IoT se faz presente como uma das grandes áreas de inovação em tecnologia. Através dela, é possível compartilhar informações sobre o uso de dispotivos pequenos, considerados pervasivos, ou seja, que estão presentes no cotidiano das pessoas sem serem percebidos. A computação em nuvem tornou-se um componente-chave para desenvolver aplicativos de IoT. Enquanto o número de dispositivos aumenta, uma grande quantidade de dados é gerada. Desse modo, é necessário tratar adequadamente o armazenamento e acesso desses dados de maneira otimizada. Existem diferentes formas de gerenciar e armazenar dados em IoT, entre elas, a abordagem relacional (SQL), o armazenamento não-relacional (NoSQL), NewSQL e modelos de larga escala. Em IoT, é comum lidar com dados não estruturados, em sua maioria, além das aplicações demandarem alta flexibilidade e mudanças frequentes de esquema. Para sistemas que demandam estes tipos de requisitos, estudos prévios indicam o uso do armazenamento não relacional, conhecido como NoSQL. Bancos de dados destes tipos flexibilizam restrições de consistência e integridade, provêem escalonamento horizontal e mecanismos otimizados de replicação e acesso. Contudo, possivelmente cada dado em IoT demande um melhor tipo de armazenamento NoSQL específico dado um critério escolhido. Partindo-se do princípio que o desempenho de leitura e escrita seja o principal critério para escolher o tipo de armazenamento, foi necessário identificar quais bancos de dados possuem o melhor desempenho dessas operações para dados de IoT, sendo este um dos objetivos do trabalho. Contudo, antes da análise experimental, fez-se necessário projetar um modelo para mapear o fluxo de dados em IoT e o direcionamento desses dados para o local de armazenamento adequado. Para isso, o presente trabalho propôs uma arquitetura de armazenamento híbrido para IoT. Afim de validá-la, experimentos utilizando os 3 principais bancos de dados NoSQL foram feitos para avaliar o tempo de inserção e leitura de dados escalares, multimídia e posicionais, 3 dos principais tipos de dados em IoT. Após os experimentos, foi identificado que o banco de dados Redis, do tipo chave-valor, obteve o melhor desempenho para escrita e consulta destes tipos de dados.

Palavras-chave: IoT, NoSQL, armazenamento, bancos de dados, armazenamento em IoT.

Abstract

IoT is present as one of the great areas of innovation in technology. Through it, it is possible to share information about the use of small devices, considered pervasive, that is, that are present in the daily lives of people without being perceived. Cloud computing has become a key component in developing IoT applications. As the number of devices increases, a large amount of data is generated. In this way, it is necessary to adequately treat the storage and access of this data in an optimized way. There are different ways of managing and storing data in IoT, including relational approach (SQL), non-relational storage (NoSQL), NewSQL, and large-scale models. In IoT, it is common to deal with unstructured data, most of which, besides applications, require high flexibility and frequent schema changes. For systems that require these types of requirements, previous studies indicate the use of non-relational storage, known as NoSQL. Databases of these types relax consistency and integrity constraints, provide horizontal scale, and optimize replication and access mechanisms. However, each data in IoT possibly requires a better type of specific NoSQL storage given a chosen criterion. Assuming that reading and writing performance is the main criterion for choosing the type of storage, it was necessary to identify which databases have the best performance of these operations for IoT data, which is one of the objectives of the study. However, prior to the evaluation, it was necessary to design a model to map the flow of data in IoT and the targeting of that data to the appropriate storage location. For this, the present work proposed a hybrid storage architecture for IoT. In order to validate it, an evaluation using the 3 main NoSQL databases were done to evaluate the insertion time and reading of scalar, multimedia and positional data, 3 of the main data types in IoT. After the evaluation, it was identified that the Redis database, of the key-value type, obtained the best performance for writing and consulting these types of data.

Keywords: IoT, NoSQL, storage, databases, IoT storage.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACID	Atomicidade, Consistência, Isolamento, Durabilidade
API	*Application Programming Interface*
BASE	*Basic Availability, soft-state*
BSON	*Binary JSON*
BLE	*BlueTooth Low Energy*
CQL	*Cassandra Query Language*
CQL	*Cypher Query Language*
HDFS	*Hadoop File System*
HD	*Hard Disk*
IDS	*Intrusion Detection Systems*
IoT	*Internet of Things*
M2M	*Machine to Machine (Máquina a Máquina)*
MQTT	*Message Queue Telemetry Transport*
MVCC	*MultiVersion Concurrency Control*
O2O	*Online to Offline*
OLAP	*Online Analytical Process*
OLTP	*Online Transactional Processing*
NoSQL	*Not only SQL*
RFID	*Radio Frequency Identification*
RSSF	Redes de Sensores sem Fio
RSL	Revisão Sistemática da Literatura
SQL	*Structured Query Language*
URI	*Uniform Resource Identifier*
XLS	*Excel Binary File Format*
XML	*Extensible Markup Language*

Sumário

1 Introdução

Nesta nova era da informática, uma das soluções que promove uma mudança de paradigma, onde os componentes são pervasivos, é a *Internet of Things* (IoT). Em IoT, grandes quantidades de dados são geradas. Nesses ambientes, o uso de uma arquitetura em nuvem é um item fundamental para o seu desenvolvimento. Entre as vantagens de se ter uma arquitetura em nuvem está a conexão de dispositivos usufruindo da escalabilidade e divisão de custos associados da infraestrutura em nuvem; provisão de armazenamento escalável, tempo de processamento ajustável e outras ferramentas para construir novos negócios (GUBBI et al., 2013).

Com a previsão de 50 bilhões de dispositivos interconectados até 2020 (GANTZ; REINSEL, 2011), o gerenciamento de uma grande quantidade de dados de forma otimizada se fará necessário. Com isso, um grande desafio levantado é a questão de como gerenciar e armazenar esses dados de forma que atenda aos requisitos das aplicações em tempo real, pois é importante frisar que o desempenho dessas aplicações estará diretamente relacionado à forma como esses dados serão controlados. Este é um problema já existente em IoT, já que, geralmente, os dispositivos possuem uma baixa capacidade de armazenamento; e numa arquitetura em nuvem, o problema toma maiores proporções. Por isso, é necessário que esses dados sejam persistidos e recuperados de forma inteligente de algum modo (AAZAM et al., 2014).

Uma das soluções que vem sendo usadas para gerenciamento de dados em demandas de Big Data são os bancos de dados usados em aplicações de computação em nuvem; nesse cenário, bancos de dados NoSQL vem se mostrando como uma boa alternativa (BOTTA et al., 2016). Esse tipo de armazenamento foi desenvolvido para trabalhar com dados não estruturados ou semi-estruturados, que hoje representam 90% dos dados digitais, incluindo dados multimídia de diversos tipos, como fotos, vídeos, mensagens de texto, etc. (GANTZ; REINSEL, 2011).

É de conhecimento geral que o modelo relacional, com sua estrutura composta de tabelas, e regras rígidas de consistência, incluindo as propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID), representa um dos modelos mais comuns e usados para armazenamento de dados nos dias atuais (STONEBRAKER, 2010). Contudo, vários questionamentos são realizados em relação a se esse modelo representa a melhor estratégia de armazenamento para aplicações de IoT. Pela rigidez de exigir dados estruturados, e pelos dados dessas novas aplicações serem, em geral, semi ou não estruturados, o modelo relacional pode não representar uma escolha adequada para esses tipos de aplicações que demandam por alta flexibilidade, escalabilidade e disponibilidade (PHAN; NURMINEN; FRANCESCO, 2014).

1.1 Problemática e Hipótese

Em aplicações de IoT existem diversos tipos de dados de sensores. Os principais tipos são: dados escalares, dados multimídia e dados posicionais (PHAN; NURMINEN; FRANCESCO, 2014). Com isso, um grande questionamento a ser realizado é como gerenciar, a um custo razoável, a grande quantidade de dados que são gerados pelos dispositivos, considerando suas especificidades.

Existem várias formas de armazenar dados de IoT. O armazenamento relacional com SQL (STONEBRAKER, 2010), NewSQL (ASLETT, 2011; CETINTEMEL et al., 2014) e, sistemas de arquivos em larga escala baseados em nuvem (ZHAO et al., 2014; BESSANI et al., 2014), são exemplos. Mesmo com a existência destas tecnologias, estudos indicam que o armazenamento NoSQL possui alta flexibilidade e disponibilidade ao lidar com a heterogeneidade de dados não estruturais, que são os tipos de dados comuns existentes em IoT (FOWLER, 2015; GROLINGER et al., 2013; AGENDA, 2016).

Bases de dados SQL geralmente escalam suas aplicações de forma vertical, ou seja, adicionando mais *hardware* aos seus servidores (ou os substituindo). Escalar verticalmente uma aplicação pode demandar um alto custo, ainda mais em aplicações de IoT. Bases de dados NoSQL foram projetadas para trabalhar com aplicações que necessitam de requisitos de alta disponibilidade e escalabilidade, e usam como estratégia a formação de *clusters* com nós que usam *hardware* comum - onde cada nó é responsável por uma parte do processamento da aplicação -, como forma de escalar suas aplicações. Esse método de escalar horizontalmente reduz custos com o aumento da demanda, se comparado com a estratégia vertical.

Diante do cenário apresentado, estudos indicam que o armazenamento em NoSQL se mostra como uma boa opção para aplicações de IoT em nuvem. Contudo, existem quatro categorias principais de bases NoSQL, que são: bancos de dados de documentos, bancos chave-valor, bancos de famílias de colunas e bancos baseados em grafos (FOWLER, 2015). Foi necessária uma análise destes modelos a fim de identificar qual ou quais se adequam melhor para dados em IoT. Com isso, o problema tratado nessa dissertação reside na necessidade de se definir um modelo de armazenamento de dados em IoT. Para isso, foi definida uma arquitetura híbrida de armazenamento NoSQL. Com esta, proveu-se o armazenamento dos tipos de dados trabalhos de forma transparente, escondendo para as aplicações demandantes os aspectos técnicos envolvidos. Para validar a arquitetura, foram realizados experimentos a fim de identificar, para dados escalares, multimídia e posicionais, qual o tipo de armazenamento NoSQL se mostra mais adequado para cada um destes, com base no desempenho de leitura e escrita, considerando que este é o principal critério para a escolha do tipo de armazenamento. Realizados os experimentos, constatou-se que o banco de dados Redis obteve melhor desempenho de leitura e escrita para os tipos de dados trabalhados.

1.2 Objetivos

Para a realização do trabalho, foram elencados um objetivo geral e uma série de objetivos específicos que serviram como um mapa para a realização das atividades. Além da definição da arquitetura, entre as atividades fundamentais estava o correto mapeamento das ferramentas necessárias para configuração do ambiente de validação da arquitetura, além da obtenção da massa de dados associada aos experimentos.

1.2.1 Objetivo Geral

Definição de uma arquitetura de gerenciamento de dados de IoT em bases NoSQL.

1.2.2 Objetivos Específicos

- Identificação dos principais tipos de dados de sensores de IoT.
- Identificação dos principais modelos de armazenamento NoSQL.
- Definição de uma arquitetura de alto nível para gerenciar dados escalares, multimídia e posicionais de sensores de IoT utilizando NoSQL.
- Realização de experimentos em relação ao desempenho de inserção e leitura dos dados trabalhados para validar a arquitetura definida.

1.3 Justificativa

Na Introdução deste trabalho, foi visto que a forma como os dados gerados em aplicações de IoT são gerenciados representa um desafio e, portanto, ainda é um problema. Diante disto, alguns trabalhos relacionados propuseram arquiteturas e outros realizaram experimentos de inserção e recuperação de dados utilizando NoSQL.

Vanelli, et al. (VANELLI et al., 2017) propuseram um *framework* de armazenamento de dados IoT em nuvem para armazenar dados escalares de sensores que compreende três camadas: *Sensores, Rede e Aplicação*. A camada de *Sensores* é responsável por coletar dados IoT e contém uma rede ZigBee e sensores, como o sensor de temperatura DHT11. A camada de *Rede* possui *gateways* que transformam dados de sensores em solicitações para a camada de aplicação. A camada de *Aplicação* recebe solicitações dos *gateways* e armazena os dados IoT no banco de dados NoSQL do AzureDB. Kebaili, et al. (KEBAILI et al., 2016) propuseram uma arquitetura para monitorar deslizamentos de terra que contém quatro camadas, a saber, *Aquisição, Transmissão, Armazenamento e Apresentação*. A camada de *Aquisição* inclui sensores para coletar dados escalares, como precipitação, umidade e um acelerômetro, para medir a velocidade dos deslizamentos. A camada de *Transmissão* obtém dados dos sensores. A camada

de *Armazenamento* utiliza o MongoDB, banco de dados de documentos NoSQL, para armazenar os dados dos sensores. A camada de *Apresentação* fornece dados aos usuários. Zhang, et al. (ZHANG et al., 2014) realizaram experimentos para validar um *framework* de armazenamento proposto por eles chamado de *HBaseSpatial*. O objetivo foi medir o tempo médio de consulta de dados espaciais do *framework* utilizando MongoDB e MySQL, banco de dados SQL tradicional. Francesco, et al. (FRANCESCO et al., 2012) realizaram experimentos para validar seu próprio *framework*, medindo o tempo médio de inserção de dados escalares e multimídia de sensores de IoT. Eles usaram um banco de dados de documentos NoSQL, o CouchDB, para medir os tempos de inserção. Os dados foram coletados por sua própria estrutura que compreende sensores integrados em um protótipo *BeagleBone*. Phan et al. (PHAN; NURMINEN; FRANCESCO, 2014) realizaram experimentos para medir o tempo médio de inserção e consulta de dados escalares e multimídia de sensores de IoT. Eles realizaram os testes através de três bancos de dados NoSQL, um banco de dados de valor-chave, a saber, o Redis, e dois bancos de dados de documentos, o CouchDB e o MongoDB. Eles também testaram usando MySQL, banco de dados SQL. Em geral, o Redis superou os outros bancos de dados testados. Zhang, et al. (ZHANG et al., 2013) realizaram uma avaliação para medir o tempo médio de consulta de dados escalares, ou seja, temperatura, umidade e concentração de dióxido de carbono usando dois bancos de dados NoSQL, Redis e HBase. Eles também utilizaram o Oracle, um banco de dados SQL. Seus testes apontam que Redis superou os outros bancos de dados.

Foram mostrados vários trabalhos anteriores relacionados ao proposto, onde arquiteturas e experimentos para validar o desempenho de aplicações utilizando modelos NoSQL foram realizados. Diante dos trabalhos anteriores apresentados, percebe-se que o tema é recorrente entre diversos pesquisadores e que várias questões ainda precisam ser resolvidas, como por exemplo, a definição de quais tipos de armazenamento possuem o melhor desempenho para leitura de dados multimídia, dados escalares e dados posicionais em IoT.

Apesar de vários trabalhos anteriores ainda realizarem testes com outros tipos de armazenamento, como com bases do tipo SQL, estudos indicam que o armazenamento NoSQL se mostra mais adequado para lidar com dados de IoT. Contudo, como cada tipo de dado em IoT demanda uma forma específica de gerenciamento, é necessário validar qual a melhor forma de gerenciar cada tipo. Após justificar o porquê de usar o armazenamento NoSQL, o próximo passo realizado foi a definição da arquitetura de armazenamento, com seus devidos componentes. Em seguida, foi realizada uma comparação das características da mesma com as arquiteturas já desenvolvidas e, por fim, uma validação da arquitetura através de experimentos.

1.4 Contribuição

Os resultados deste trabalho promovem um *background* de experimentos que servem de base para novas pesquisas acerca do armanezanamento de dados em IoT, a criação de aplicações

mais confiáveis, além da possibilidade de melhorias nas tecnologias de armazenamento existentes. Além disso, promove as seguintes contribuições:

- Identifica qual tipo de armazenamento NoSQL é o mais indicado para trabalhar com cada tipo de dado de sensor proposto, considerando o melhor custo/benefício. Esse quesito será obtido após a realização dos experimentos com cada tipo de armazenamento para cada tipo de dado, onde será considerado como de melhor retorno aquele que apresentar melhor resultado em relação aos itens levantados nos casos de testes, ou seja, o tipo de armazenamento que se destacar no maior número de características, será elencado como de melhor custo/benefício. Uma consequência direta disso é o início do uso massivo das arquiteturas adequadas para os requisitos demandados, fazendo com que se diminua o risco de o desenvolvedor de aplicações consumidoras implementar sua solução com um tipo de armazenamento que não é adequado para o seu contexto.

- Identifica quais tipos não são indicados para essas aplicações e as razões. Esse quesito será obtido após a execução dos experimentos com a obtenção dos esquemas de armazenamento que apresentarem o pior resultado em relação aos itens levantados nos casos de testes, considerando cada tipo de armazenamento que teve pior retorno para cada tipo de dado. Como serão avaliadas 4 arquiteturas de armazenamento com três tipos distintos de dados, poderão haver casos em que uma ou mais dessas arquiteturas de armazenamento não proverá desempenho satisfatório para um ou mais desses tipos de dados de sensores testados.

- Devido ao teor dos experimentos que foram realizados, desenvolvedores de futuras aplicações consumidoras tem acesso a técnicas de configuração e uso dos bancos de dados trabalhados, onde estes podem se aproveitar dessas vantagens fornecidas, auxiliando na melhora do desempenho tanto na leitura quanto na escrita dos dados em suas aplicações.

- Partindo-se do princípio de não haver uma solução de armazenamento perfeita para o problema levantado, após o experimento, foram identificados os pontos fortes e fracos de cada abordagem para cada tipo de dados testado. Com isso, é possível aos desenvolvedores escolher modelos que mais se adequem às suas demandas prioritárias. Com isso, haverá casos que desempenho na leitura de dados seja um requisito prioritário sobre a escrita de dados ou vice-versa.

- Define uma arquitetura híbrida de alto nível, com um modelo para o armazenamento escalável em IoT contendo camadas com diferentes funções, como a coleta dos dados e a correta identificação do melhor local para armazenamento de cada tipo de dado. Essa arquitetura serve como base para que novas aplicações possam ser corretamente endereçadas, aproveitando-se de melhores modos de armazenamentos destacados nos resultados dos experimentos do trabalho.

1.5 Estrutura do Documento

A proposta de dissertação foi organizada em cinco capítulos que serão distribuídos da seguinte forma:

- O capítulo 1 apresenta o problema, a justificativa e a proposta da dissertação.
- O capítulo 2 aborda os principais temas e tecnologias do trabalho.
- O capítulo 3 descreve os trabalhos relacionados, suas contribuições e limitações.
- O capítulo 4 contém a metodologia e os procedimentos metodológicos utilizados no trabalho.
- O capítulo 5 contém a arquitetura proposta e suas características.
- O capítulo 6 contém os resultados experimentais.
- O capítulo 7 contém a conclusão e os trabalhos futuros.

2 Referencial Teórico

Nesta seção são descritos conceitos essenciais previstos na literatura para os assuntos relacionados com o trabalho, que são: A Internet das Coisas, a Computação em Nuvem e o armazenamento NoSQL, com detalhes dos seus tipos e exemplos. Essas explicações tem como objetivo nortear o leitor no entendimento das principais características destas tecnologias, a fim de prover uma base teórica necessária para o entendimento do trabalho proposto.

2.1 Internet das Coisas

O termo *Internet of Things* (IoT), definido por Kevin Ashton (ASHTON, 2009), se baseia na ideia de fazer com que qualquer coisa seja endereçada e ingresse na Internet. Para que seja possível implementar a IoT, é necessária uma mudança de visão de componentes móveis e portáveis para componentes embutidos e imperceptíveis no cotidiano. É preciso entender o contexto do usuário e a aplicação desses elementos pervasivos de comportamento autônomo. Nesse âmbito, a próxima revolução é interconectar objetos formando um ambiente inteligente, onde é preciso administrar de forma segura, eficiente e escalável os dados gerados na rede. IoT representa a interconexão entre dispositivos em um *framework* unificado, compartilhando informações entre diferentes plataformas (GUBBI et al., 2013).

Os dispositivos inteligentes de aplicações em IoT são entidades que seguem algumas características, entre elas: possuem forma física, detém um conjunto mínimo de funcionalidades de comunicação; estão associados a pelo menos um nome ou endereço; possuem identificação única na rede; possuem algum processamento computacional básico, como por exemplo a capacidade de casar padrões de mensagens de entrada como nas *tags* RFID, ou até mesmo realizar serviços de descoberta e gerenciamento de tarefas de rede; podem possuir sensibilidade a fenômenos físicos, como aumento de temperatura, radiação, etc., ou realizar ações com efeito físico, como no caso dos atuadores (MIORANDI et al., 2012).

2.1.1 Características chave de dispositivos e aplicações em IoT

Uma característica existente em IoT é a heterogeneidade dos seus dispositivos, aplicada tanto a nível do *hardware* empregado, como nas arquiteturas e protocolos existentes. Devido a falta de padronização, é comum que as aplicações necessitem implementar *middlewares* para comunicação e troca de dados e informações de dispositivos até mesmo com função similar. A escalabilidade representa uma segunda característica marcante, onde a comunicação dos dispositivos está diretamente ligada a capacidade das aplicações implementarem serviços de descoberta de nomes e endereços dos dispositivos, de trocar e interconectar dados, e de gerenciar

os dados e conhecimento adquirido entre os dispositivos interconectados. Uma tecnologia muito comum nos dias atuais para troca de dados são as redes *wireless*. Ter dispositivos pervasivos utilizando os recursos dessas redes representa uma característica também marcante, pois os dispositivos podem usufruir de uma infraestrutura já existente para alavancar o desenvolvimento das aplicações consumidoras (MIORANDI et al., 2012).

Outra característica fundamental existente são as aplicações de uso consciente de energia, principalmente se for considerado que a autonomia desses dispositivos em aplicações ainda é baixa. Por serem muito pequenos (pervasivos), a maioria dos dispositivos de IoT possuem baixa autonomia energética, o que leva a criação de soluções para uso de forma consciente da capacidade energética do dispositivo, a fim de aumentar a vida útil do mesmo. Uma outra frente se concentra em tentar desenvolver novas tecnologias de *hardware* com direcionamento a diminuir cada vez mais o tamanho dos dispositivos e, em contrapartida, aumentar a sua autonomia energética. A existência de funcionalidades de rastreamento e localização é outro ponto marcante, onde áreas como a de logística podem usufruir dessa capacidade para implementar soluções das mais diversas. Um outro ponto representa a capacidade autonômica que vários desses dispositivos possuem. Partindo-se do cenário comum que os dispositivos devem poder operar sem intervenção humana pelo maior tempo possível, é comum se ter esses equipamentos em aplicações de IoT onde estes possuem múltiplas operações e funcionalidade de contingência no caso de problemas.

2.1.2 Tecnologias usadas em IoT

Algumas tecnologias que são usadas na construção das aplicações em IoT são: as redes de sensores sem fio (RSSF), tags RFID (*Radio-Frequency Identification)*, serviços em nuvem e M2M *(Machine-to-Machine)* (ROMAN; ZHOU; LOPEZ, 2013). Com IoT, é possível criar soluções em vários domínios de aplicação, como o automotivo, saúde, logística, etc. As redes de sensores de sem fio (RSSF) têm exercido um papel fundamental na implementação e execução de aplicações em IoT, devido a possibilidade de os dispositivos poderem utilizar da infraestrutura já existente nestas redes para trocar dados, realizar o reconhecimento e identificação de novos dispositivos, entre outras atividades. RFID constitui uma das principais tecnologias utilizada em IoT, exercendo um papel fundamental na identificação de dispositivos em aplicações. Contudo, devido a heterogeneidade entre os dispositivos, citada anteriormente, algumas características do RFID faz que alguns problemas surjam, como por exemplo ser utilizado apenas para identificação de dispositivos e rastreamento de objetos embarcados (MIORANDI et al., 2012).

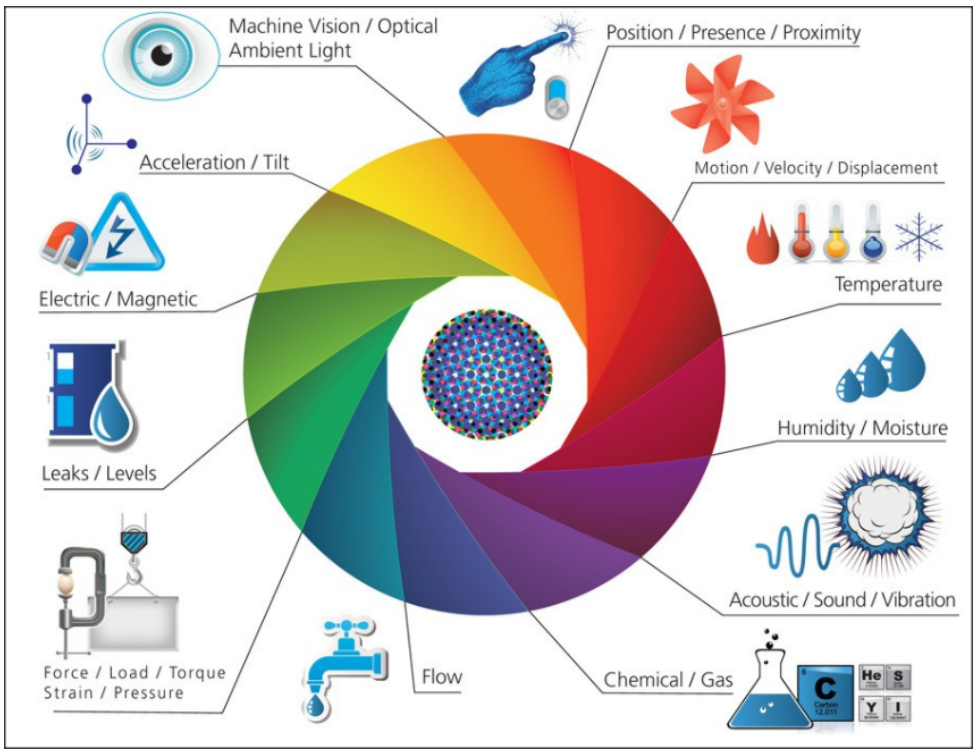
Quando considerado que soluções em IoT bem estruturadas estão dispostas em forma de camadas com funções bem definidas, a computação em nuvem tem exercido um papel importante como parte de uma dessas camadas. Com ela, é possível usufruir de armazenamento escalável e, com isso, poder utilizar serviços de reconhecimento e gerenciamento de dispositivos utilizando recursos que estão em nuvem e não apenas dependendo da capacidade de armazenamento dos dispositivos (WORTMANN; FLÜCHTER et al., 2015).

2.1.3 Tipos de sensores de IoT

Em IoT existem diversos tipos de sensores, cada um com sua função principal, mas também é possível que um sensor acumule mais de uma função, como por exemplo, ser ao mesmo tempo um sensor de temperatura e de umidade. Em geral, existem diversos tipos de sensores, entre eles: sensores de proximidade, de aceleração, de direção, de temperatura, de pressão, de umidade, de nivelamento, etc.

Os sensores de proximidade detectam movimentações em seu entorno e geralmente são utilizados para realizar alguma ação quando algo se aproxima deste. Por exemplo, é possível ter um dispositivo, com uma câmera e um sensor destes acoplado, que envie *snapshots* para um sistema de segurança sempre que algum objeto ou pessoa se aproximar dele. Já os sensores de aceleração detectam vibrações, inclinação e a aceleração linear. Exemplos de uso: dispositivos anti-roubo e pedômetro (serve para contar passos, monitorar distância percorrida, etc.). Entre os sensores de direção, pode-se citar seu uso em dispositivos do tipo giroscópio, que servem para ajustar a direção e a orientação de veículos, aeronaves, etc (WORLD, 2018). Sensores de temperatura podem ser utilizados na climatização de ambientes. Sensores de pressão podem ser usados, por exemplo, na calibração de pneus em fábricas. Sensores de umidade por ser utilizados em meteorologia e os sensores de nivelamento podem ser utilizados em diferentes contextos, como ajuste no nível de líquidos e combustíveis, no controle de irrigação, etc. (RCRWIRELESSNEWS, 2018). A Figura 1, proposta em (POSTSCAPES, 2018), mostra diversos tipos de sensores utilizados em IoT.

Figura 1 – Exemplos de sensores de IoT (POSTSCAPES, 2018)

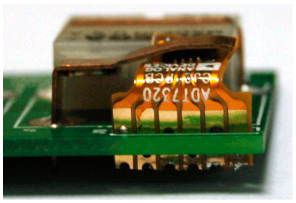


2.1.4 Dispositivos e sensores de IoT

Existem diferentes dispositivos e sensores que são utilizados para aplicações de IoT. É importante frisar que, normalmente, um sensor é uma parte de um dispositivo, onde este pode desempenhar outras funções além da que já é executada pelo seu sensor acoplado. Os sensores mais utilizados em aplicações de IoT são sensores de dados escalares (de grandeza numérica). Dentre estes, tipos comuns são os sensores de temperatura, pressão e umidade. Um exemplo de sensor de temperatura é o ADT7320 (DEVICES, 2018a), que representa um componente pequeno que pode ser acoplado a placas e outros dispositivos, como por exemplo junto ao circuito CN0172 (DEVICES, 2018b). Em relação a aferição de pressão, um exemplo de sensor deste tipo é o LP25H (ST, 2018), que representa um componente muito utilizado em barômetros, medidores pressão de pneus, etc. Já em relação ao item umidade, um exemplo de sensor deste tipo é o HTU21D (SPARKFUN, 2018), que representa um componente que pode ser utilizado para verificar o nível de umidade em solos para aplicações de agricultura ou mesmo para avaliar se há risco de deslizamento de terra em alguma área de risco. A Figura 2a representa o circuito CN0172 com o sensor de temperatura ADT7320, a Figura 2b representa o sensor de pressão LP25H e a Figura 2c representa o sensor de umidade HTU21D.

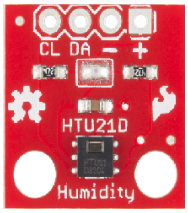
Figura 2 – Sensores de temperatura (ADT7320), pressão (LPS25H) e umidade (HTU21D)

(a) ADT7320



(b) LP25H

(c) HTU21D



2.2 Computação em Nuvem

A Computação em Nuvem surgiu a partir da necessidade de se fornecer níveis de serviços em diferentes camadas, onde o usuário pode usufruir da vantagem de poder solicitar ao provedor algum destes serviços que o mesmo não possua ou não tenha condições de possuir. Os serviços fornecidos podem ser de *hardware*, *software*, virtualização, armazenamento, etc., onde cada um destes é precificado e fornecido por diversos provedores distintos, o que leva o usuário a ter uma gama maior de opções de contratação e níveis de serviços.

Outras vantagens de se ter um modelo terceirizado de fornecimento de serviços que podem ser comprados são: baixo custo de operação para o usuário, já que a responsabilidade pela manutenção do serviço fica a cargo do provedor; provisão de alta escalabilidade, já que, a depender do nível de crescimento do usuário, o mesmo pode rapidamente solicitar a adição de novos componentes aos já contratados; fácil acesso, pois os serviços estão disponíveis pelo

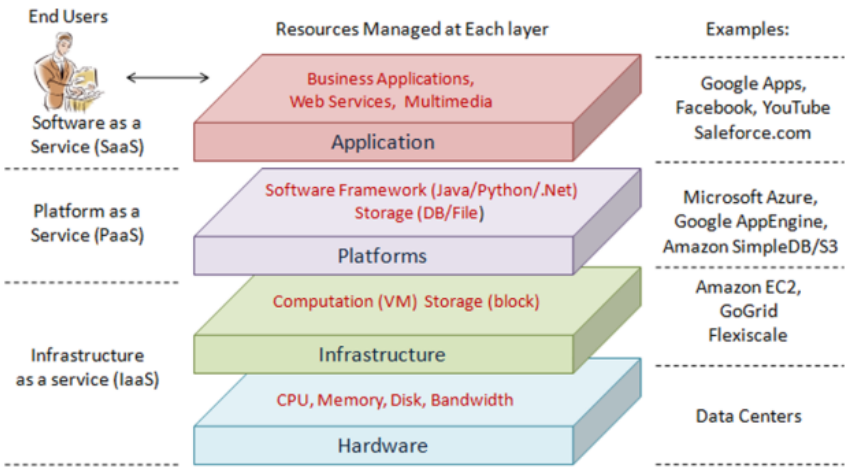
terceiro, onde a responsabilidade por manter o acesso ao serviço é do provedor, e ainda, no contrato firmado com o usuário, o provedor precisa manter o serviço disponível sob pena de multas e sanções.

2.2.1 Conceitos chave

Segundo o NIST (NIST, 2018), Computação em Nuvem é um modelo para acesso por demanda via rede a um repositório com recursos computacionais compartilhados (servidores, rede, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e entregues com o mínimo de esforço do provedor do serviço. Partindo-se de uma arquitetura conceitual em camadas, a Computação em Nuvem pode ser dividida em 3 classificações: Infraestrutura como Serviço *(Infrastructure as a Service, IaaS)*, Plataforma como Serviço *(Platform as a Service, PaaS)* e Software como Serviço *(Software as a Service, SaaS)*. Infraestrutura como Serviço representa a camada mais baixa e se refere a provisão por demanda de recursos de infraestrutura em forma de serviços, normalmente sob a forma de virtualização (ZHANG; CHENG; BOUTABA, 2010).

Como exemplos de provedores de IaaS temos a Amazon EC2 (AMAZON, 2018a), GoGrid (GOGRID, 2018) e Flexiscale (FLEXISCALE, 2018). Plataforma como serviço se refere a provisão de recursos em forma de plataformas, como por exemplo suporte a sistemas operacionais e *frameworks* de desenvolvimento de aplicações. Como exemplos de provedores de PaaS temos Google App Engine (GOOGLE, 2018a), Microsoft Windows Azure (WINDOWS, 2018). Já a última classificação, de *Software* como Serviço, se refere a provisão sob demanda de aplicações, pela Internet, como serviços. Exemplos de provedores de SaaS incluem Salesforce.com (SALESFORCE, 2018), Rackspace (RACKSPACE, 2018b) e SAP Business ByDesign (SAP, 2018). A Figura 3 representa graficamente os conceitos explicados.

Figura 3 – Arquitetura em Camadas da Computação em Nuvem (ZHANG; CHENG; BOUTABA, 2010)



2.2.2 Vantagens da Computação em Nuvem

Uma das maiores vantagens presentes na computação em nuvem é poder prover elasticidade e alta escalabilidade de uma forma que os usuários não necessitem investir mais e mais recursos em *hardware*, novos servidores, e *software*. Isso é feito através da provisão de entrega de componentes e aplicações como serviços, onde o usuário não precisa adquirir produtos que não precise, e, com isso, ele pagará apenas por aquilo que ele realmente vai utilizar, que é a forma como são chamadas as nuvens públicas. Com isso, o serviço é vendido seguindo um modelo de computação utilitária. Esses serviços são os classificados na camada de *Software as a Service (SaaS)*, mencionada anteriormente. Já quando se têm uma nuvem que é apenas disponível para uma organização e seus funcionários, esta é chamada de nuvem privada.

Do ponto de vista de *hardware*, têm-se 3 características chave que a Computação em Nuvem fornece: primeiramente, tem-se a ilusão de que os recursos são infinitos, já que o usuário não mais necessitará ter posse de todo o *hardware* e *software* para seus produtos, uma vez que estes componentes estão de posse de outros fornecedores; em segundo lugar, elimina-se também a necessidade de o usuário ter que gastar em equipamentos e infra-estrutura que vão além da sua necessidade, com isso, o usuário pode começar pequeno e aumentar o seu nível de uso à medida que for necessário. E por último, será eliminada a necessidade de o usuário ter que gastar em equipamentos que poderiam passar muito tempo sem uso devido a falta de necessidade ou capacidade do negócio do consumidor. Além das características mencionadas, o usuário irá reduzir custos com energia, operações, equipamentos, sistemas e funcionários. Com isso, pode-se dizer que as soluções de computação em nuvem costumam ser mais baratas a possuir *datacenters* próprios, além de ter desempenho muitas vezes equiparável (FOX et al., 2009).

2.2.3 *Frameworks* de Computação em Nuvem

Na definição de ambientes em nuvem, considerando o modelo arquitetural explicado anteriormente, a camada de IaaS representa um componente primordial na estrutura, devido a ser nela que a virtualização é configurada e implementada. A virtualização representa uma funcionalidade essencial na configuração de ambientes em nuvem, pois com ela é possível configurar e utilizar serviços de diferentes tipos a um baixo custo, além de ser mais simples identificar erros e prover uma infraestrutura mais tolerante a falhas. Diante da importância da camada de IaaS e da virtualização, alguns *frameworks* possuem funcionalidades chave para definição de infraestruturas completas de Computação em Nuvem (MORENO-VOZMEDIANO; MONTERO; LLORENTE, 2012).

O OpenNebula (OPENNEBULA, 2018) representa uma plataforma em nuvem IaaS de livre licença utilizada para gerenciar *data centers* virtualizados com nuvens privadas, públicas e híbridas. O grande objetivo da plataforma é prover funcionalidades para abstrair recursos como rede, armazenamento, virtualização, monitoramento e gerenciamento de usuários. Por ser uma ferramenta com controle centralizado, todos os componentes da plataforma são controlados pelo

componente *Cloud OS*, que tem o papel de gerenciar a virtualização de recursos, provisionando os mesmos sempre que necessário (MORENO-VOZMEDIANO; MONTERO; LLORENTE, 2012). A plataforma funciona sob o paradigma de mestre/escravo, onde o *front-end* de acesso e gerenciamento administrativo representa o nó mestre e todos os demais nós da arquitetura são considerados escravos que rodam máquinas virtuais. Em relação ao armazenamento, existem dois tipos de dados, imagens e arquivos, que são compartilhados através da rede pelo nó mestre (*front-end*) para os nós escravos.

Uma segunda ferramenta, com função análoga ao OpenNebula é a OpenStack (RACKSPACE, 2018a). O OpenStack, diferente do OpenNebula, que possui uma arquitetura centralizada, possui uma série de subcomponentes relacionados entre si que devem ser configurados juntos para a implantação da ferramenta. Com isso, a arquitetura do OpenStack demanda que cada subcomponente seja instalado em um nó diferente, onde estes são inicializados juntos na arquitetura. Uma segunda diferença, é que o OpenStack é direcionado para nuvens privadas. O OpenStack possui 4 serviços principais: *Dashboard*, que representa um *web front-end* onde o administrador pode interagir com o nós e configurar recursos; *Compute*, que é responsável por gerenciar e controlar toda a plataforma; *Networking* é o recurso que possui a entrega de serviços de conectividade, balanceamento de carga e *firewall*; já o componente *Storage* é divido nos tipos *Object*, que são os dados persistentes, tolerantes a falhas escaláveis e *Block*, que representa o armazenamento em bloco, normalmente utilizado para cenários de alta dependência do desempenho.

2.3 Integração entre Internet das Coisas e Computação em Nuvem

Para um efetivo desenvolvimento das aplicações de IoT, a Computação em Nuvem se tornou um forte aliado. Entre as vantagens de aliar as duas tecnologias, temos a conexão de dispositivos usufruindo da escalabilidade e divisão de custos associados da infraestrutura em nuvem; provisão de armazenamento escalável, tempo de processamento ajustável, ferramentas para construção de novos negócios; além de dar a opção de o usuário pagar apenas pelo que usar, que é uma das características fundamentais das aplicações nas nuvens (GUBBI et al., 2013).

Dispositivos de IoT, especialmente os embarcados, possuem algumas limitações, como por exemplo, a baixa capacidade de armazenamento e processamento dos dados. Esse problema acaba tomando proporções maiores, à medida que o número de dispositivos que se integram à rede aumenta. O uso da Computação em Nuvem para compor o *testbed* das aplicações de IoT faz com que esses problemas sejam escalados para o nível da nuvem, tornando viável a solução destes problemas.

Na integração entre IoT e *Cloud*, vários são os desafios que ainda se encontram sem solução, entre eles: a definição de padrões de protocolos e ontologias que serão utilizados para

facilitar a unificação na comunicação dos dispositivos, o controle do consumo energético destes dispositivos, o uso em larga escala do protocolo IPV6, tratamento de qualidade de serviço (QoS), além de questões como segurança e armazenamento dos dados trafegados (AAZAM et al., 2014). Entre os desafios citados, neste trabalho será tratado o armazenamento e gerenciamento de dados em aplicações de IoT.

Existem vários projetos de pesquisa e produtos empresariais destinados a prover ambientes de intregação entre IoT e *Cloud*. OpenIoT (OPENIOT, 2018) representa um *middleware* de livre licença que tem como proposta obter informações de dispositivos como sensores e atuadores e fornecer serviços de IoT com dados destes dispositivos em nuvem. A ideia de oferecer serviços de sensores e atuadores em nuvem gerou uma nova camada, conhecida como S2aaS (*Sensing-as-a-Service*). O Nimbits (NIMBITS, 2018), outra plataforma de livre licença, objetiva conectar coisas à nuvem. Possui 2 componentes centrais: um *web server*, responsável por processar dados geográficos e temporais, além de executar regras do usuário para gerenciar dados e dar notificações; e uma biblioteca Java para conexão e integração de novas aplicações na plataforma, além de prover uma biblioteca para conexão com Arduino, com um aplicativo Android para gerenciamento dos recursos. A troca de dados é feita utilizando o padrão *JSON* (DÍAZ; MARTÍN; RUBIO, 2016).

2.3.1 Armazenamento em aplicações de IoT em *Cloud*

Aplicações de IoT normalmente são compostas de sensores e atuadores, como citado anteriormente. Esses dispositivos geram uma infinidade de dados e necessitam de alta disponibilidade para trocar informações entre si de forma performática. Uma questão é como gerenciar e armazenar os dados na integração entre IoT e *Cloud Computing*. O problema de armazenamento neste contexto também já existe em aplicações de IoT que não utilizam *Cloud Computing*, onde os dispositivos possuem uma baixa capacidade de armazenamento de dados, como citado anteriormente. Com isso, unir essas aplicações em um ambiente em nuvem faz com que o gerenciamento dessas informações trocadas seja mais sofisticado.

Igualmente ao problema da eficiência energética, vários grupos de dispositivos de IoT em nuvem fará o problema ficar maior, já que a quantidade de dados trafegados será muito maior, e esses dados precisarão ser persistidos de forma inteligente de algum modo. Uma outra questão, que é crucial nessas simulações, é que os dados sejam sempre retornados quando demandado, já que um dispositivo pode precisar acessar um dado presente em outro no mesmo momento em que pode estar havendo uma atualização de informações no dispositivo requisitado.

Para o armazenamento em aplicações de IoT em *Cloud* algumas abordagens são possíveis. Uma delas é utilizando armazenamento SQL, onde poderá haver uma maior latência no retorno do dado requisitado, já que, como bases relacionais valorizam consistência, o dado apenas seria retornado após o fim de uma atualização. NewSQL e Sistema de arquivos de larga escala baseados em nuvems seriam outras abordagens. Uma última abordagem seria a adotada por bases NoSQL,

que retornariam o dado no momento da solicitação, independentemente de o mesmo estar num processo de atualização ou não; já que em bases NoSQL, a prioridade é a disponibilidade e não a consistência (PHAN; NURMINEN; FRANCESCO, 2014; FOWLER, 2015). Com isso, como o armazenamento em IoT demanda por alta disponibilidade e flexibilidade, estudos indicam que é mais adequado utilizar armazenamento NoSQL.

2.4 Bases de Dados NoSQL

As bases de dados NoSQL são assim denominadas por seguirem um modelo do tipo BASE *(Basic Availability, Soft-state, eventual consistency)*, que prega basicamente que o mais importante não é que o esquema de armazenamento seja estritamente consistente e disponível, e sim que esses itens sejam disponibilizados de forma menos arbitrária. Em contra partida, o sistema é mais tolerante a falhas. Diversas são as características, vantagens e desvantagens que existem ao se trabalhar com esse modelo de armazenamento. Esses pontos serão tratados a seguir.

2.4.1 Diferenças básicas entre SQL e NoSQL

Uma das diferenças principais entre bases SQL e NoSQL é que as últimas escalam bem à medida que a demanda aumenta, pregando um modelo onde as aplicações são escaladas com a formação de uma rede de máquinas de baixo custo processando de forma distribuída, ao contrário de aplicações com armazenamento SQL que geralmente escalam incrementando ou mesmo substituindo o *hardware* dos servidores. Outra diferença é que o modelo relacional trabalha com uma estrutura fixa de tabelas com linhas e colunas, enquanto em NoSQL é possível se ter tabelas com várias colunas, onde estas são usadas por várias linhas; também é possível existir tabelas de atributos; ter vários relacionamentos muitos para muitos; possuir características de árvore; além de requisitar mudanças de esquema frequentes, por lidar frequentemente com dados não estruturados (FOWLER, 2015; GROLINGER et al., 2013).

Um ponto que deve ser levado em consideração é que bases SQL trabalham bem com dados estruturados, e, grande parte dos dados gerados e tratados por diversas aplicações, como em redes sociais, por exemplo, são dados não estruturados. Dados não estruturados não são preparados para trabalharem com a estrutura padrão de tabelas, linhas e colunas da forma que as bases SQL trabalham, o que pode causar um trabalho maior em realizar essa adaptação a preferir implementar o armazenamento da aplicação diretamente em uma base com arquitetura NoSQL, além de em muitos casos as bases SQL serem vinculadas a funcionalidades e estruturas mais complexas e desnecessárias à aplicação, o que normalmente não acontece quando o armazenamento utiliza um modelo NoSQL. Outro complicador é que bases SQL não foram projetadas para particionar dados. Com isso, realizar junção de tabelas de forma distribuída é uma tarefa difícil de ser realizada (LEAVITT, 2010).

2.4.2 Vantagens em se utilizar NoSQL

Existem algumas vantagens em se utilizar bases NoSQL. A primeira delas é que normalmente as bases NoSQL possuem melhor desempenho que as em SQL. Em grande parte, isso se deve ao fato de as bases NoSQL não implementarem as restrições ACID implementadas em SQL, que degradam bastante o desempenho da aplicação objetivando ter alta consistência e integridade dos dados. A atomicidade prega que uma transação não pode ser realizada em partes, ou seja, ou ela é completamente realizada ou não. Consistência se refere que nenhuma transação pode quebrar regras do banco, já o isolamento se trata da capacidade de cada transação ser executada independentemente de outras em execução, e, por fim, durabilidade se refere que transações completas serão persistidas. Todas essas características citadas são flexibilizadas no modelo não relacional, ou seja, podem ser implementadas ou não. Com isso, a maioria das bases NoSQL são simples de serem entendidas e usadas por desenvolvedores.

2.4.3 Desvantagens em se utilizar NoSQL

Também existem uma série de desvantagens em se utilizar bases NoSQL. A primeira delas é que como a maioria delas não trabalha com SQL, as consultas a dados devem ser também criadas pelo desenvolvedor da aplicação, o que pode demandar grande esforço caso se trabalhe com consultas de dados complexas e com muitos relacionamentos. Outro fator preponderante seria a confiabilidade de se trabalhar com essas bases, já que, como elas não implementam ACID, restrições desde as mais simples são completamente relaxadas, o que pode ser um problema, caso a aplicação demande o mínimo de consistência e integridade. Um outro ponto que deve ser levado em consideração é a falta de familiaridade no uso deste tipo de armazenamento. Como existem diferentes modelos e arquiteturas, muitas vezes a equipe não sabe qual o modelo é mais indicado para demanda da sua aplicação. Outro fator é que muitas dessas bases, por não terem a mesma disseminação de bases SQL, não possuem o devido suporte para apoio na implantação e manutenção de sistemas em produção.

2.4.4 Tipos de Bancos de Dados NoSQL

Existem diferentes tipos de arquiteturas de bancos de dados não relacionais, desde bancos hierárquicos, até mesmo bases orientadas a objeto. Partindo-se do princípio de que cada uma possui vantagens e desvantagens, e diante da dúvida de qual tipo de arquitetura NoSQL atenderia bem para cada tipo de dado em IoT, serão trabalhados os quatro tipos arquiteturais principais, que são: bancos chave-valor, bancos de dados de documentos, bancos de famílias de colunas e bancos baseados em grafos (FOWLER, 2015; TIWARI, 2011).

2.4.4.1 Bancos de dados chave-valor

Representam o tipo mais simples, onde basicamente armazenam os dados seguindo um modelo de chave única e valor, onde *hash maps* podem ser definidos para esses pares de valores. Os bancos deste tipo armazenam e fazem todo o processamento utilizando normalmente cache de memória. Estas bases possuem esquema completamente livre, onde qualquer dado pode ser inserido a qualquer momento sem interferir com outros dados já existentes, além de ser necessário que o relacionamento entre os componentes seja tratado na lógica da aplicação. Apesar de utilizar memória, o que indica baixa capacidade de armazenamento em algumas situações, esses modelos contam com replicação, versionamento, bloqueio de operações, transações, etc. (HECHT; JABLONSKI, 2011)

Em relação aos *clients* desse tipo, temos como exemplos os bancos Redis (REDIS, 2018), Voldemort (VOLDEMORT, 2018), Riak (TECHNOLOGIES, 2018), Tokyo Cabinet (LABS, 2018) e Scalaris (ZIB, 2018). Uma característica interessante de todos esses *clients* é a capacidade de prover escalabilidade através da distribuição das chaves entre os nós. Outro ponto é que, geralmente, as interfaces de usuários desses *clients* provêm inserções, deleções e consultas com uso de índices. Vale ressaltar que nenhum dos bancos desse tipo oferecem índices secundários.

Alguns desses *clients*, como Redis, Voldemort, Riak e Tokyo Cabinet, armazenam dados na memória ou em HD. Já os outros bancos armazenam na memória, mas utilizam o espaço de HD como *backup* ou já contam de forma nativa com replicação e recuperação, o que faz com que *backup* não seja estritamente necessário. Por padrão, os bancos desse tipo utilizam replicação de forma assíncrona, com excessão do banco Scalaris. Outra característica interessante é a capacidade de versionar utilizando um controle de concorrência de múltiplas versões, como no caso dos *clients* Voldemort e Riak. Contudo, os bancos que não usam essa funcionalidade fazem bloqueio de versionamento, para evitar inconsistências (CATTELL, 2011).

2.4.4.2 Bancos de Dados de Documentos

Esses tipos de bases de dados, diferentemente do tipo chave-valor, têm a característica de armazenar quaisquer tipos de objetos, os quais são chamados de "documentos". Utilizam *JSON* para organizar pares contendo um identificador único do documento e o documento em si. Suportam qualquer tipo de dado, são completamente livres de esquema e também possuem múltiplos atributos para consulta dos registros, o que dá muita liberdade para o desenvolvedor das aplicações consumidoras, além de facilitar tarefas de integração de dados e migração de esquemas. Suportam índices secundários, para aumentar o desempenho de consultas, além de permitir objetos mistos, em alguns modelos, e listas.

O uso deste tipo de banco de dados tem algumas vantagens, como, por exemplo: documentos são tipos nativos em muitas linguagens de programação; o uso de documentos de forma recursiva e em listas reduz a necessidade de muitas junções, que, quando usadas nas

bases relacionais, degradam o desempenho da aplicação; utilização de esquemas dinâmicos que suportam o uso de polimorfismo (MONGODB, 2018b). Outras propriedades interessantes desses bancos de dados são a capacidade de realizar consultas a coleções baseado no valor de múltiplos atributos, não prover bloqueios de transações explicitamente, além de flexibilizar concorrência e as propriedades ACID. Em relação aos *clients* de bancos de dados de documentos, temos como exemplos os bancos MongoDB (MONGODB, 2018b), SimpleDB (AMAZON, 2018b), CouchDB (APACHE, 2018f) e Terrastore (GOOGLE, 2018d)

Os *clients* de bancos de dados de documentos utilizam arquiteturas bem próximas, mas terminologias diferenciadas. Um exemplo disso está no conceito de coleção usado no MongoDB, que no SimpleDB é tratado como domínio, o mesmo conceito é tratado como banco de dados no CouchDB e como *bucket* no Terrastore. Os "documentos"do MongoDB são chamados de itens no SimpleDB e atributos do MongoDB são chamados de campos no CouchDB. Apesar dessas pequenas diferenças semânticas, o funcionamento é análogo, como mencionado (CATTELL, 2011).

2.4.4.3 Bancos de dados de famílias de colunas

Representam um modelo que trabalha com linhas e colunas, onde estas são divididas em múltiplos nós para fornecer escalabilidade com uso de funções *hash*, fazendo com que as consultas utilizando intervalos não sejam obrigadas a percorrer os nós um a um. Linhas são divididas entre os nós fragmentando a chave primária, já as colunas são distribuídas entre os nós formando grupos de colunas objetivando deixar que colunas mais relacionadas fiquem juntas, devido a alguma necessidade de negócio.

Nessas bases, as linhas seguem um princípio análogo ao das bases de documentos, onde o nome da linha deve ser único. As colunas e linhas podem ser adicionadas de forma muito flexível em tempo de execução, mas normalmente as colunas têm que ser pré-definidas, o que acaba deixando estes tipos menos flexíveis que bancos de documentos e chave-valor. As colunas de uma tabela também são distribuídas entre os nós utilizando grupos de colunas. Isso pode causar um aumento na complexidade, mas, em contrapartida, traz a vantagem, já mencionada, de deixar colunas relacionadas juntas (CATTELL, 2011).

Entre os *clients* desse tipo têm-se como exemplos os bancos BigTable (GOOGLE, 2018b), HBase (APACHE, 2018c), HyperTable (HYPERTABLE, 2018) e Cassandra (APACHE, 2018a). Os *clients* HBase e BigTable trabalham com o conceito mencionado de famílias de colunas. O *client* Cassandra, diferente dos anteriores, trabalha com uma terceira dimensão, chamada de super colunas, que representa uma variação do conceito de famílias de colunas, onde estas podem conter colunas ou "super colunas". Além dessas características, como o Cassandra possui controle de concorrência com múltiplas versões, possui maior flexibilidade para tratar dados mais complexos e estruturas de dados mais expressivas, ao contrário de bancos como HBase e HyperTable, que focam em consistência forte com bloqueios e *log* das operações. No Cassandra,

como os valores não são interpretados pelo sistema, os relacionamentos entre os conjuntos de dados não são fornecidos nativamente, devendo o desenvolvedor da aplicação, caso necessite, implementar esta lógica. Este tipo de banco de dados trabalha bem com grandes quantidades de dados com esparsos *clusters*, porque os dados podem ser particionados de forma eficiente.

2.4.4.4 Bancos de dados baseados em grafos

Representam um modelo que segue a teoria dos grafos, onde trabalham-se com nós e arestas, e as bases de dados desse tipo lidam com objetos armazenados que são fortemente inter-relacionados, como por exemplo quando se quer descobrir o relacionamento entre pessoas em uma rede social. Os nós e arestas possuem pares de chave e valor embutidos, e com isso podem ser definidos esquemas onde é possível que arestas sejam aplicadas apenas a um tipo específico de nó (VICKNAIR et al., 2010). Representam o único modelo entre os principais que é preocupado com relacionamentos e o foco na representação visual da informação, de forma que esse tipo se torna mais "amigável"para os humanos entre os tipos NoSQL.

Uma vantagem desse modelo é que operações transversais reduzem o custo de junções recursivas que precisariam ser realizadas em outros modelos. RDF representa um tipo especial de bases de grafos onde, apesar de não ser possível adicionar pares de chave-valor adicionais para nós e arestas, permite definir tipos de dados mais complexos e esquemas mais completos do que nas bases de grafos comuns. Entre os *clients* desse tipo têm-se os bancos Neo4j (NEO4J, 2018), Titan (TITAN, 2018), FlockDB (TWITTER, 2018) e GraphDB (ONTOTEXT, 2018). De certa forma é possível considerar um grafo como uma das estruturas mais úteis para modelar objetos e suas interações.

O Twitter é um exemplo de aplicação que utiliza bases de grafos para definir os relacionamentos entre seus participantes, onde é utilizado um *client* próprio do grupo, por nome de FlockDB (TWITTER, 2018), que possui uma otimização para lista adjacentes muito grandes, leituras e escritas rápidas. Este *client* é aplicável, com alta escalabilidade, para relacionamentos com 1 salto entre os nós.

2.5 Bancos de Dados NoSQL (Ferramentas)

Nesta seção serão mostrados exemplos de bancos de dados NoSQL dos principais tipos previstos, além de suas arquiteturas e características técnicas. Serão mostrados dois exemplos de bancos de dados NoSQL para cada tipo. O critério utilizado para escolha das bases foi o seu nível de uso segundo o *ranking* de *engines* de bancos de dados mais utilizados (IT, 2018).

2.5.1 Redis

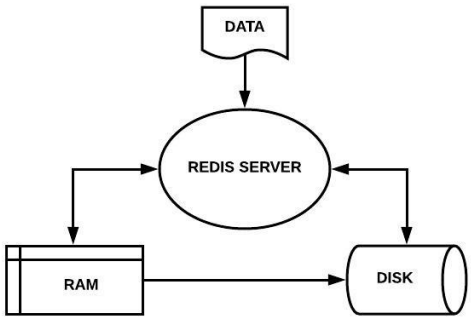
O Redis (REDIS, 2018) representa o banco de dados chave-valor mais utilizado. A chave para identificação dos registros utiliza funções de *hash* consistente ao invés da identificação

utilizando intervalos de chaves, como no MongoDB, onde blocos de chaves dos intervalos são armazenados em nós separados, o que facilita consultas de conjuntos de dados, já que chaves vizinhas são replicadas em vários servidores, mas pode deixar o *cluster* dos nós mais complexo.

O uso de *hash* para identificar registros no Redis facilita a localização de registros com o cálculo dessas funções, o que atenua a necessidade de balanceamento de carga, existente na identificação por intervalos (HECHT; JABLONSKI, 2011). Possui como uma de suas características, igualmente ao CouchDB, oferecer consistência eventual de forma nativa. Outra característica do Redis é o uso de *locking* otimizado de transações, atenuando a perda de desempenho da aplicação quando é essencial que um dado seja gravado antes de lido, num caso destas operações ocorrerem de forma paralela.

O Redis armazena todos os dados e suas variações como *strings*. Coleções, listas, *maps* são todos formados por *strings* e existe um estrutura especial denominada de *string dinâmica* ou SDS, formada por 3 componentes, que são: *buff, len e free*. O primeiro representa uma lista de caracteres que armazena a *string*; o segundo é um número que armazena o tamanho da lista e o último é a quantidade de *bytes* adicionais que podem ser utilizados. Os dados com Redis são mantidos na memória primária, mas podem ser persistidos no HD quando necessário. Possui um sistema interno próprio para gerenciamento da memória, onde, quando um valor é trocado com o HD, um ponteiro para a página do HD é armazenado com chave. Além do gerenciador de memória virtual, o Redis possui bibliotecas para gerenciar operações via *socket*. O Redis não necessita de um sistema operacional para realizar *swapping* porque os objetos não são mapeados um por um com paginação. Quando uma porcentagem mínima de objetos são acessados, é possível que vários arquivos de paginação sejam acessados simultaneamente. Ao contrário do MongoDB, no Redis, os dados em HD não são os mesmos que em memória. Os dados em HD no Redis são comprimidos, com isso, a paginação customizada diminui a entrada e saída de dados em HD (TIWARI, 2011). Com o Redis é possível gravar os dados apenas em memória, apenas em disco ou em memória e posteriormente em disco. A Figura 4 representa a arquitetura do Redis.

Figura 4 – Arquitetura do Redis



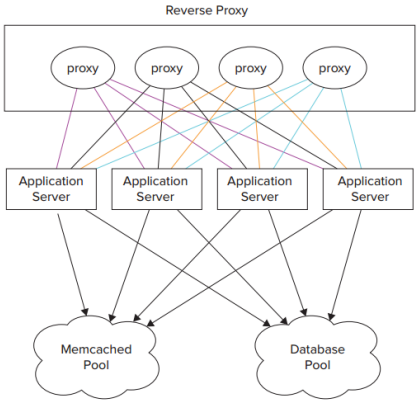
2.5.2 Memcached

O Memcached (MEMCACHED, 2018) é o segundo banco chave-valor mais utilizado, onde um servidor contendo essa base é considerado como uma grande tabela *hash*. Possui poucas restrições, como por exemplo: não faz *backup* e nem recuperação de falhas, não lida com listas, coleções, além de não fazer persistência. Por isso, em alguns casos, não é considerado como uma base de dados. A função básica do Memcached em uma aplicação é reduzir a carga de armazenamento e, para isso, utiliza apenas um único índice para todos os dados. Igualmente ao Redis não suporta índices secundários. Apesar das limitações, é bastante aproveitado e utilizado por grandes empresas como Youtube e Facebook (CATTELL, 2011).

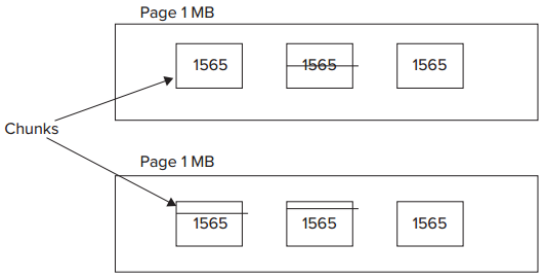
O componente central do Memcached é denomidado de *slab allocator*, onde cada componente de armazenamento pode conter pequenos espaços, conhecidos como *chunks* ou pedaços maiores de armazenamento, que são os *buckets*. Ter tamanhos de memória variados pode causar desperdício de espaço, além de caches de memória antigos, o que é evitado com o uso de algoritmos de troca de página, que vão descartando páginas mais velhas e deixando apenas as mais recentes. Além disso, a realocação de memória também pode ser utilizada para minimizar esse problema. Vale ressaltar que o uso do Memcached é mais indicado para aplicações web. As Figuras 5a e 5b representam o modelo de funcionamento do Memcached (TIWARI, 2011).

Figura 5 – Memcached numa aplicação web e *chunks*

(a) Memcached numa aplicação web



(b) *Chunks* no Memcached



2.5.3 MongoDB

O MongoDB (MONGODB, 2018b) representa o banco de dados do tipo documento mais utilizado. Quando se agrupam os documentos juntos formam-se coleções. As coleções podem ser conceitualmente comparadas com as tabelas do modelo relacional, com a diferença que não possuem por padrão as restrições que existem nas tabelas do modelo relacional. Também é possível que todos os documentos do banco sejam colocados juntos formando uma única coleção. Um ponto que deve ser ressaltado é que os documentos de uma coleção devem ser

relacionados para facilitar a indexação posterior em consultas. Coleções podem ser separadas através de *namespaces*.

Um documento no MongoDB é armazenado no formato BSON, que representa uma codificação em binário do formato *JSON*, que é definido por uma estrutura de conjuntos de pares chave-valor. O BSON é uma estrutura evoluida do *JSON* que permite o uso de expressões regulares, dados binários e datas. cada documento possui um identificador único que é gerado automaticamente pelo MongoDB, caso o desenvolvedor não explicite o identificador no momento de adicionar um documento numa coleção. Em aplicações que utilizam MongoDB, os dados em BSON são serializados e deserializados para poderem ser lidos. Contudo, no servidor do MongoDB os dados não necessitam de serialização, pois são entendidos.

MongoDB é um banco que preza por alto desempenho, prova disso é o uso massivo de arquivos de mapeamento de memória, que são segmentos de memória virtual que são validados *byte* a *byte* em relação a um descritor de arquivo. Com isso, as aplicações podem acessar esses dados como se eles tivessem diretamente na memória primária, o que aumenta consideravelmente o desempenho da aplicação. Com isso, acessar e manipular a memória é muito mais rápido que fazer chamadas ao sistema operacional. Apesar das vantagens desse modelo, existem alguns problemas. Primeiro, Não existe uma clara separação entre o *cache* do banco e o *cache* do sistema. Segundo, todo o *cache* é controlado pelo sistema, porque a memória virtual trabalha da mesma forma em todos os nós. O que quer dizer que quando não há a possibilidade de definir políticas de *cache* separadas, as regras são completamente mantidas ou completamente descartadas entre os nós.

Como citado anteriormente, o MongoDB não aplica por padrão as restrições ACID das bases relacionais, as quais podem ser adicionadas separadamente quando necessário. Outro ponto é que o MongoDB escreve no disco a cada minuto e qualquer falha entre duas sincronizações, de escrita e atualização, por exemplo, gerará inconsistência nos dados. Se for necessário um aumento na consistência, podem ser adicionadas escritas no disco mais frequentes, apenas ressaltando que isso degradará o desempenho da aplicação. Esse é um ponto que deve ser sempre balanceado.

MongoDB trabalha com o conceito de fragmentação e replicação dos dados. A replicação é uma função muito útil para evitar a perda completa de dados. Normalmente se trabalha com o conceito de mestre e escravo, onde um nó é o mestre e outros são escravos. É importante destacar que a replicação dos dados ocorre de forma assíncrona, com isso, os dados podem estar desatualizados. Contudo, ainda assim é melhor utilizá-la e ter dados desatualizados a certo momento a ter uma perda completa dos dados em uma falha.

Uma outra técnica mais avançada que a replicação dos dados que também pode ser utilizada seria a fragmentação. A ideia básica da fragmentação é dividir os dados entre os nós em fragmentos. Com isso, cada nó representa um fragmento de uma coleção, o que promove uma estratégia de escalonamento de forma horizontal. Além disso, a fragmentação e a replicação

normalmente são utilizadas juntas, onde se fragmentam os dados de uma coleção entre alguns nós e replicam-se os dados desses fragmentos em outros nós. Nas versões mais recentes do MongoDB a replicação e a fragmentação já são implementadas nativamente (TIWARI, 2011).

A Figura 6a representa a estrutura de um documento no MongoDB e a Figura 6b representa o modelo de fragmentação no MongoDB.

Figura 6 – Estrutura de um documento no MongoDB e fragmentação

(a) Estrutura de documento no MongoDB

(b) Fragmentação no MongoDB

O MongoDB possui uma rica linguagem de consulta que suporta operações de CRUD das aplicações, como, por exemplo, um *framework* para agregação de dados, outro para pesquisas em texto e outro para consultas geoespaciais.

2.5.4 CouchDB

O CouchDB (APACHE, 2018f) representa o segundo tipo de banco de documentos mais utilizado que, como o MongoDB, trabalha com os conceitos de documentos e coleções. As coleções são compostas apenas dos esquemas, e índices secundários devem ser criados explicitamente em campos nessas coleções. Já um documento pode conter atributos simples, como dados numéricos, textuais, booleanos ou atributos compostos como outros documentos ou listas. No tocante às consultas, no CouchDB elas são chamadas de "visões"e são definidas utilizando a linguagem *Javascript*. Os índices são implementados utilizando árvores B, que são úteis para escalar grandes quantidades de dados e permitir rápida recuperação destes. Com isso, os resultados das consultas possuem uma estrutura ordenada ou de valores em intervalos. Um ponto interessante é que as consultas podem ser distribuídas entre os nós utilizando o mecanismo de *map reduce*.

Para proteção contra perdas de dados acidentais, em suas versões iniciais e diferentemente do MongoDB, o CouchDB escalava suas aplicações utilizando apenas replicação assíncrona, pois não continha nenhum mecanismo para realizar fragmentação. Contudo, após o projeto Lounge (GOOGLE, 2018c), as versões mais recentes já implementam fragmentação entre os nós. Outra característica é que provê durabilidade para aplicações gravando em disco todas

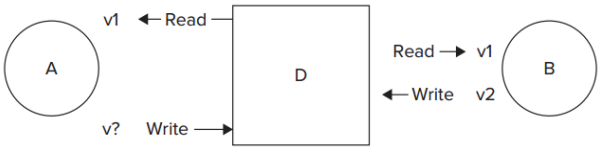
as atualizações após o *commit*. Um outro ponto é que os desenvolvedores de aplicações têm disponível *APIs* de diversas linguagens de programação para chamada ao CouchDB no formato *RESTful*. Essas *APIs* têm a função de converter chamadas nativas do MongoDB em requisições *RESTful* nas aplicações (CATTELL, 2011).

O CouchDB possui algumas diferenças arquiteturais em relação ao MongoDB. A primeira diferença é que a identificação dos documentos no CouchDB é feita através do uso de *hashing* consistente, enquanto no MongoDB os documentos são identificados com o particionamento dos ID's desses documentos. Um segundo ponto é que o CouchDB trabalha com o conceito de consistência eventual de forma nativa com o intuito de aumentar a disponibilidade, onde o desenvolvedor pode definir o nível de consistência dos nós, regulando questões como permitir que um mesmo dado seja lido enquanto ocorre uma a atualização neste, ao mesmo tempo, onde o valor retornado será o antigo até o momento que a atualização termine. No CouchDB é priorizada a alta disponibilidade em relação à consistência dos dados. É importante destacar que nem toda aplicação se encaixa bem nesse conceito, como, por exemplo, aplicações bancárias; onde é necessário que o dado lido seja sempre o de sua versão final, o que demanda alta consistência. Uma terceira característica do CouchBD é o uso de replicação no modelo mestre-mestre, onde, em um esquema de vários nós com replicação de dados, onde um deles é o líder, caso o líder falhe, outro nó componente se torna mestre de forma automática. Essa característica faz com que o CouchBD trabalhe bem com aplicações *offline*, como no caso de alguns aplicativos para celular (HECHT; JABLONSKI, 2011).

Para lidar com problemas de conflitos devido ao uso da consistência evetual, o CouchDB implementa controle de concorrência com multiplas versões (MVCC), para cada documento do banco de forma individual, onde um identificador é criado de forma automática para cada documento. O uso de MVCC implica que podem ocorrer consultas e atualizações ocorrendo em paralelo sem necessidade de bloqueio de transações, além de implicar também que todas as escritas são sequenciadas utilizando uma propriedade *"_rev"* que serve para manter a versão mais recente de um atributo. Com isso, os dados retornados pelo cliente podem ser sempre os mais recentes. Contudo, vale ressaltar que em atualizações simultâneas, o CouchDB notifica a aplicação quando um documento está recebendo múltiplas atualizações desde o momento que ele foi obtido. Contudo, fica a cargo do desenvolvedor definir uma lógica na aplicação para combinar as atualizações ou sobrescrever a anterior com a mais recente. A Figura 7 mostra um exemplo de possíveis conflitos, evitados pelo MVCC, que podem ocorrer quando dois clientes, A e B, estão lendo um mesmo dado em paralelo e vão fazer atualização do dado de forma simultânea. (TIWARI, 2011)

O MVCC controla o versionamento das escritas, como mencionado, fazendo com que o cliente faça uma releitura do dado antes da nova atualização, para que ele saiba que o dado foi atualizado no momento em que ele solicitou nova atualização, deixando a cargo do cliente decidir se a atualização deve ocorrer ou não. Com isso, evita conflitos de atualização entre os

Figura 7 – Possível conflito evitado pelo uso do MVCC no CouchDB



clientes.

2.5.5 Cassandra

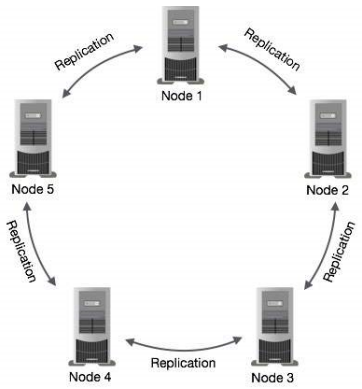
O Cassandra (APACHE, 2018a) representa o banco de famílias de colunas mais utilizado. As bases, chamadas de *keyspaces*, são formadas por grupos de colunas, onde se tem o conceito de supercoluna, que promove um novo nível de agrupamento a um grupo de colunas. Uma família de colunas pode conter supercolunas ou colunas, nunca os dois em conjunto.

Em bases Cassandra, ao se formar um *cluster* com os nós, todos estes possuem o mesmo papel, podendo receber e solicitar requisições de leitura e escrita. Um dado interessante sobre a arquitetura do Cassandra é que quando é identificado que um nó retornou um dado desatualizado para um cliente, o dado mais recente também é enviado, logo em seguida. Também ocorre que no momento seguinte o nó que tinha a informação desatualizada é atualizado (CASSANDRA, 2018).

O Cassandra trabalha com operações de inserção, atualização, deleção e alteração de esquema utilizando a linguagem *CQL*, juntamente com uma *API* em Java. Os dados das operações ficam armazenados em cache na memória e são gravados em disco posteriormente, onde estes dados no disco são periodicamente compactados. Promove escalabilidade elástica, onde não há um ponto único de falha, além de possuir a característica de poder implementar as propriedades ACID, dos bancos de dados relacionais. Cassandra trabalha com um esquema de replicação em anel, onde os dados são fragmentados entre os nós para impedir um ponto único de falha, como representado na Figura 8. A replicação ocorre de forma otimizada, com uso de fragmentação dos dados, onde os fragmentos são gravados entre diversos nós de forma distribuída. Contudo, é importante ressaltar que o Cassandra oferece um modelo de controle de concorrência mais fraco que em outras bases, pois não possui mecanismos de bloqueio de transações e a replicação dos dados ocorre de forma assíncrona (CATTELL, 2011).

Igualmente ao Redis, utiliza funções *hash* para identificar seus registros ao realizar o particionamento entre nós de forma horizontal. O uso do *hash* de forma ordenada no Cassandra oferece benefícios semelhantes aos dos índices em árvores B. Contudo, realizar classificação e ordenamentos dos dados geralmente ocorre de forma mais lenta que em modelos que usam árvores B.

Figura 8 – Replicação no Cassandra



2.5.6 HBase

O HBase (APACHE, 2018c) representa o segundo banco de famílias de colunas mais utilizado. Utiliza o HDFS, que representa um sistema que provê o armazenamento de grandes arquivos distribuídos, com intuito de fornecer tolerância a falhas. Possui suporte a recuperação de falhas automático, leituras e escritas consistentes, escalonamento de forma linear, se integra com o *Hadoop* (TUTORIALSPOINT, 2018a), tanto quanto fonte, como destino; possui uma *API* Java para aplicações e provê replicação através dos seus *clusters*.

As atualizações vão sempre para o fim de arquivos de dados, para evitar consultas inconsistentes e para facilitar a recuperação de falhas se o servidor ficar inutilizado. As operações são atômicas e realizadas sobre as linhas, que possuem bloqueio de transações. Também é possível utilizar um controle de concorrência mais forte, abortando atualizações quando surgir conflitos. Sua arquitetura possui 3 componentes principais: *MasterServer*, responsável por controlar a replicação e fragmentação dos dados nas regiões; as *Regions*, que nada mais são que tabelas divididas em regiões; e *ZooKeeper* que é um projeto para manter informações de configuração, replicação, etc. A Figura 9a representa estes componentes.

Vale ressaltar que a replicação dos dados ocorre no modelo mestre-escravo, onde se tem mais de um mestre para evitar um ponto único de falhas. Esse processo é controlado por um modelo em *MapReduce* que realiza a distribuição e replicação dos dados de forma eficiente. O HBase utiliza árvores B para permitir consultas mais velozes e classificação dos dados. A Figura 9b exemplifica a interligação desses componentes (TIWARI, 2011).

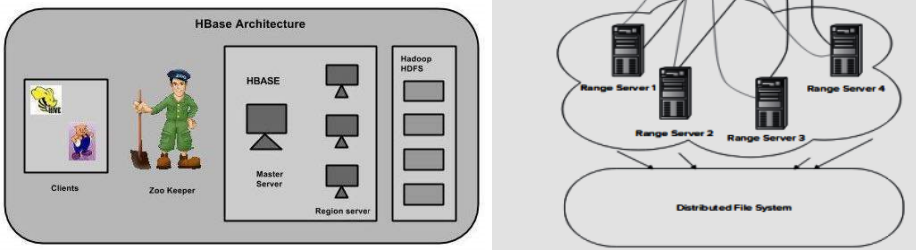
2.5.7 Neo4j

O Neo4j (NEO4J, 2018) representa o banco de dados de grafos mais utilizado. Possui o *CQL* como sua linguagem de consulta de dados entre os nós, utiliza índices para facilitar as buscas transversais entre os nós, suporta restrição de unicidade, implementa as propriedades ACID por completo, consegue exportar os dados dos nós para os formatos *JSON* e *XLS*. Além disso, possui *APIs REST* que podem ser acessadas por várias linguagens de programação. As

Figura 9 – Componentes e comunicação entre estes no HBase

(b) Comunicação no HBase

(a) Componentes do HBase

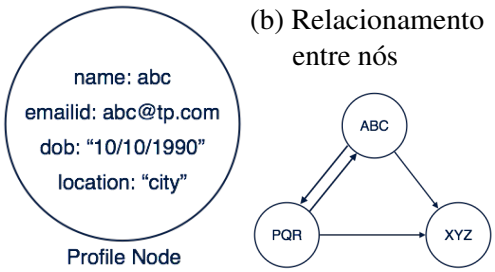


maiores vantagens do Neo4j são: facilidade e rapidez para ler dados nos nós utilizando operações transversais, utiliza dados semi-estruturados, o que dá liberdade de mudança de esquema, não requer junções complexas para consultas de dados. Como desvantagem, não provê fragmentação de dados entre os nós de forma nativa. O modelo de dados do Neo4j trabalha com nós, arestas (relacionamentos) e propriedades, onde as propriedades podem ser aplicadas tanto aos nós quanto às arestas (TUTORIALSPOINT, 2018b). As propriedades representam pares chave-valor e os relacionamentos possuem comunicação direcional ou bidirecional. A Figura 10a representa um exemplo de nó com dados de perfil de uma rede social e a Figura 10b o relacionamento com outros nós da rede.

Figura 10 – Nó de perfil em rede social e relacionamento com outros nós

(a) Exemplo de nó

(b) Relacionamento entre nós



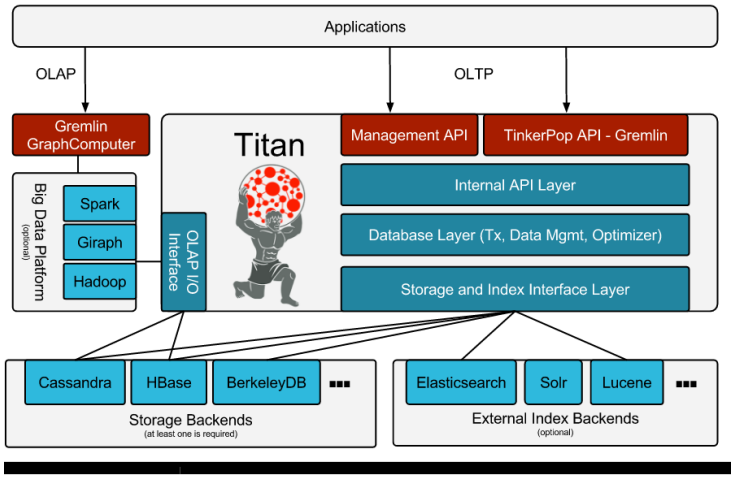
2.5.8 Titan

O Titan (TITAN, 2018) representa o segundo banco de grafos mais utilizado. Possui como características escalar de forma elástica e linear, replicação de dados, suporte a ACID, suporte a consistência eventual, suporte a várias plataformas de *Big Data* como *Spark* (APACHE, 2018d), *Giraph* (APACHE, 2018b) e *Hadoop* (TUTORIALSPOINT, 2018a).

A Figura 11 representa a arquitetura do Titan e seus componentes. A arquitetura provê suporte para aplicações transacionais (OLTP) e analíticas (OLAP). Para aplicações OLTP, a

arquitetura do Titan fornece conexão com a interface *Gremlin* do *framework TinkerPop*, o que permite utilizar transações no Titan (APACHE, 2018e). Para as aplicações OLAP, a *API* do Titan fornece comunicação com plataformas de *Big Data* citadas. Tanto as aplicações OLAP como OLTP possuem *backends* para conexão com bancos do dados (Ex.: Cassandra) e também para índices.

Figura 11 – Arquitetura do Titan e seus componentes



3 Revisão Sistemática da Literatura

Nesta seção são mostrados os resultados de uma revisão sistemática da literatura. Nela, além da definição da pergunta de pesquisa, são apresentados os objetivos da revisão, as palavras-chave definidas, critérios de inclusão e exclusão de trabalhos, a estratégica utilizada para execução da revisão e os resultados da mesma. Por fim, as características dos trabalhos selecionados na revisão sistematica são explicados nesta seção.

3.1 Definição da Pergunta de Pesquisa

Qual a estratégia ou técnica de armazenamento adequada para gerenciar operações de inserção e recuperação de dados escalares, multimídia e posicionais de Internet das Coisas?

3.2 Objetivos da Revisão Sistemática e da Pesquisa

Este estudo tem como objetivo identificar trabalhos anteriores relacionados à definição de arquiteturas de armazenamento de dados de aplicações de Internet das Coisas. O foco deste estudo é a descoberta e definição de estratégias adequadas para armazenamento e recuperação de dados de forma otimizada em IoT, considerando-se que cada tipo de dado de IoT possui sua especificidade e que é possível que cada tipo de dado, entre os escolhidos para o estudo, possua uma forma arquitetural distinta no tocante ao seu armazenamento e recuperação.

Através deste estudo, será possível mapear o estado da arte acerca do problema abordado, de forma a identificar as lacunas existentes, além de possibilitar um possível direcionamento e validação de soluções e teorias já propostas por outros pesquisadores, a fim de enriquecer a pesquisa e solucionar ou aproximar-se da solução do problema levantado.

3.3 Palavras-chave de Pesquisa

Para a correta identificação de trabalhos sobre arquiteturas de armazenamento em IoT, considerando-se modelos auxiliados pela Computação em Nuvem, utilizando as bases IEEE Xplore, ScienceDirect e Scopus, a Tabela 1 mostra o comando com suas variações por base e os totais retornados.

3.4 Critérios de Inclusão e Exclusão

Para auxiliar na filtragem de trabalhos relacionados, alguns critérios de inclusão e exclusão foram definidos. A Tabela 2 detalha os códigos para critério de inclusão e suas respectivas descrições. Já a Tabela 3 detalha os códigos para critério de exclusão e suas respectivas descrições.

3.5 Estratégia de execução da RSL

Para execução da RSL, foi adotada a seguinte estratégia: A revisão foi dividida em 3 rodadas, onde na primeira rodada o objetivo principal foi realizar um filtro de exclusão inicial dos artigos retornados. O foco da filtragem nessa rodada foi a exclusão de trabalhos que não atendam aos critérios "1E", "2E"e "3E". Na segunda rodada, após o filtro inicial realizado, o foco principal foi na verificação de artigos que não atendiam aos critérios "4E"a "10E". Já na terceira e última rodada, fase com a leitura integral dos artigos, um nova filtragem foi realizada e os trabalhos não excluídos fizeram parte das seções de trabalhos relacionados.

3.6 Resultados da RSL

O foco de execução da RSL foi na busca por proposta de arquitetura de armazenamento NoSQL para IoT, além da busca por trabalhos que trazem contexto experimental ligado a esse tema. Após a execução da RSL, a Tabela 4 mostra o quantitativo de artigos por critério de inclusão e a Tabela 5 o quantitativo de artigos por critério de exclusão.

Acerca dos bancos de dados na definição de arquiteturas da RSL, a Tabela 6 identifica as quantidades de trabalhos por banco de dados. Já em relação aos trabalhos experimentais da RSL, a Tabela 7 identifica a quantidade destes em relação a cada banco de dados.

Vale ainda destacar que 4 dos trabalhos experimentais, aproximadamente 12% de artigos relacionados da RSL (34 artigos), continha a definição de um experimento onde os sensores estão acoplados em uma *protoboard*, dos tipos: *Raspberry Pi, BeagleBone* ou *Arduino*.

Acerca dos tipos de dados de sensores nos artigos da RSL, a Tabela 8 mostra as quantidades identificadas.

Tabela 1 – Quantidade de artigos localizados por base

Base	**Comando de Busca**	**Totais**	
IEEE Xplore	(IoT OR Internet of Things) AND (storage OR database) AND Cloud AND NoSQL AND Sensor NOT "Document Title":Survey NOT "Document Title":Security NOT "Document Title":Challenge* NOT "Document Title":Issues NOT "Document Title":Perspective* NOT "Document Title":StudyNOT "Document Title":Implication*NOT "Document Title":Consideration*NOT "Document Title":Direction*	313	529**
Scopus	(IoT OR Internet of Things) AND (storage OR database) AND Cloud AND NoSQL AND Sensor AND NOT TITLE(*Survey*) AND NOT TITLE(*Security*) AND NOT TITLE(*Challenges*) AND NOT TITLE(*Issue*) AND NOT TITLE(*Perspective*) AND NOT TITLE(*Study*) AND NOT TITLE(*Implication*)AND NOT TITLE(*Consideration*) AND NOT TITLE(*Direction*)	98	
ScienceDirect	(IoT OR Internet of Things) AND (storage OR database) AND Cloud AND NoSQL AND Sensor AND NOT title(Survey) AND NOT title(Security) AND NOT title(Challenges) AND NOT title(Issues) AND NOT title(Perspectives) AND NOT title(Study) AND NOT title(Implications) AND NOT title(Considerations) AND NOT title(Directions)	148	
**	Total após remoção de duplicados.		

Tabela 2 – Critérios de inclusão de artigos da RSL

Identificador	**Critério de Inclusão**
1I	Arquitetura, estudo de caso ou *framework* para armazenamento de dados de IoT
2I	Experimentos sobre armazenamento de dados de IoT

Tabela 3 – Critérios de exclusão de artigos da RSL

Identificador	**Critério de Exclusão**
1E	Trabalho não relacionado ou com foco diverso
2E	Artigos em espanhol, japonês ou em mandarim.
3E	Falta de arquitetura, *framework* ou estudo de caso sobre armazenamento em IoT e de experimento
4E	Não menção ou subutilização de NoSQL na definição de arquitetura
5E	Não menção a NoSQL na parte experimental
6E	Não menção a Computação em Nuvem na arquitetura
7E	Não trabalha com dados escalares, multimídia ou posicionais
8E	Não contém ferramentas da dissertação
9E	Seção ou trecho relacionado à dissertação superficial ou pouco caracterizado
10E	Experimento sem sensores relacionados nem há comparações entre bancos de dados

Tabela 4 – Quantitativo de artigos por critérios de inclusão

Quantidade de Artigos (critérios de inclusão)		
Proposta de Arquitetura (1I)	**Contexto Experimental Relacionado (2I)**	**Arquitetura e Experimentos (1I e 2I)**
17	15	2

Tabela 5 – Quantitativo de artigos por critérios de exclusão

Quantidade de Artigos (critérios de exclusão)	
1E	355
2E	1
3E	62
5E	10
6E	2
7E	1
8E	28
9E	9
10E	9
Artigos indisponíveis	5

Tabela 6 – Quantitativo de banco de dados em relação a definição de arquiteturas

Quantidade de artigos arquiteturas por banco de dados	
MongoDB	7
HBase	5
Redis	1
Cassandra	1
Neo4j	0
CouchDB	0

Tabela 7 – Quantitativo de banco de dados em relação a artigos experimentais

Quantidade de artigos experimentais por banco de dados	
MongoDB	9
CouchDB	4
HBase	4
Cassandra	3
Redis	2
Neo4j	1

Tabela 8 – Quantidade de artigos por tipos de dados de sensores

Quantidade de artigos por tipos de dados de sensores	
Escalares (temperatura, umidade, pressão, etc.)	13
Multimídia (fotos, vídeos, etc.)	6
Posicionais (latitude, longitude, etc.)	5

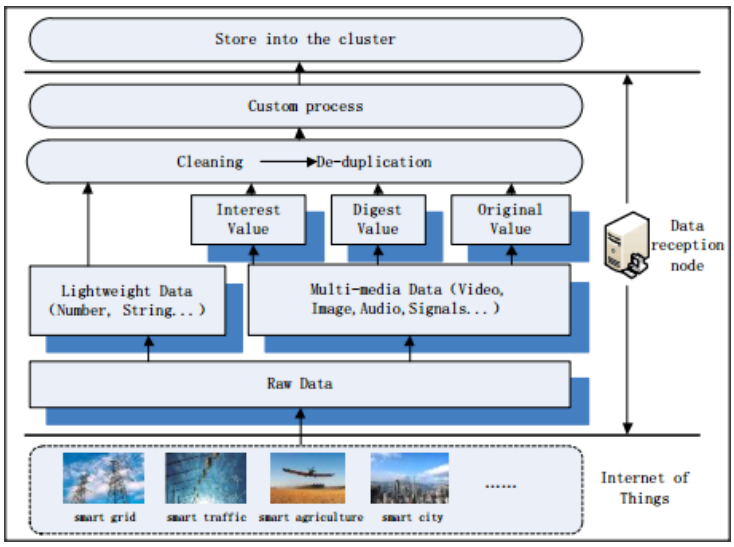
4 Pesquisa por Trabalhos Relacionados

Finalizada a escolha dos trabalhos pela realização da revisão sistemática, é chegado o momento de destacar as características de cada trabalho escolhido. Nesta seção, são descritas as características dos trabalhos com definição de arquiteturas de armazenamento em IoT, e também dos trabalhos de cunho experimental. Feito isto, ao final do capítulo está elencada um tabela comparativa com as diferenças entre os trabalhos escolhidos e também destes em relação ao trabalho realizado nessa dissertação.

4.1 Propostas de Arquitetura Relacionadas

Uma série de trabalhos possui definem uma arquitetura de armazenamento de dados de IoT. Em (LI et al., 2012), os autores defendem que bases NoSQL são mais indicadas para armazenamento em IoT devido a sua composição com dados não estruturados, em larga escala, que demandam alta disponibilidade e flexibilidade. Neste trabalho é proposta uma arquitetura, por nome de IOTMDB, com 4 tipos de nós. O primeiro e principal nó é o *master node*, que é responsável por aceitar conexões de novos clientes, gerenciar os fragmentos entre os nós, verificar se um nó atingiu a sua capacidade máxima, etc. O segundo nó é o *standby node*, que é utilizado para prevenir um ponto único de falha. Já o *data reception node* é responsável por receber dados dos sensores e realizar algum pré-processamento. Já os *slave nodes* são os nós que contêm todos os dados das aplicações. A Figura 12 representa a arquitetura.

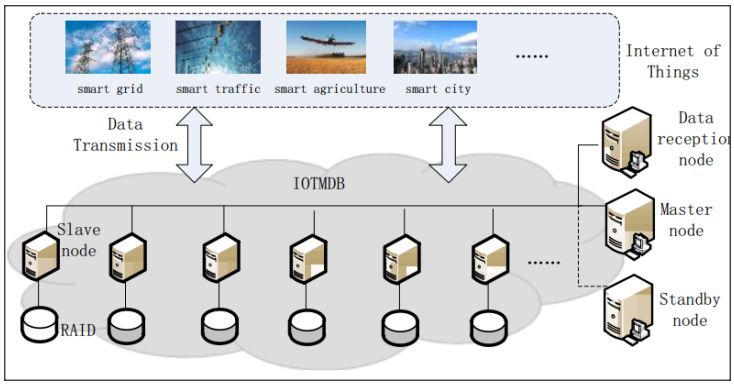
Figura 12 – Arquitetura de nós do IOTMDB



Os autores também definem um conjunto de 4 ontologias para representar a arquitetura. Entidade, que pode ser representada por temperatura, luminosidade, pressão, umidade, etc.;

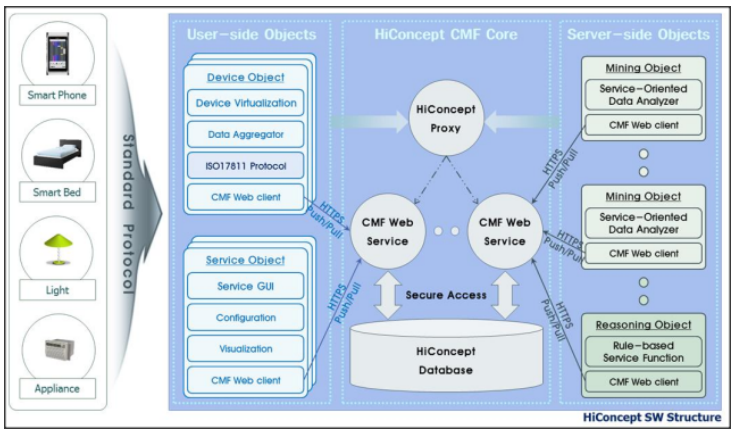
Campo, que pode variar a depender de qual o tipo da entidade; Atividade, usada para descrever atividades; e Aplicação, que pode conter diversas aplicações e entidades. Um outro aspecto relevante da arquitetura IOTMDB é referente ao pré-processamento dos dados. Os dados são divididos em duas categorias, dados leves (temperatura, pressão, umidade, etc.) e dados multimídia (fotos, vídeos, áudios, etc.). Além desta divisão, todos os dados passam por um processo de limpeza e de-duplicação, para eliminar inconsistências e duplicações desnecessárias. Além disso, os dados leves são gravados de forma separada dos dados multimídia, um dos pontos que demandam isso é a necessidade maior de espaço demandada pelos dados multimídia. A figura 13 elucida os componentes de pré-processamento dos dados.

Figura 13 – Arquitetura de pré-processamento de dados do IOTMDB



Em (SON et al., 2015), os autores definem um *framework*, chamado de HiConcept, para armazenamento de dados de sensores de temperatura utilizando o banco de dados MongoDB, usufruindo de requisições HTTP POST para capturar os dados dos sensores e gravar na base através de *JSON*. A figura 14 representa a arquitetura do HiConcept.

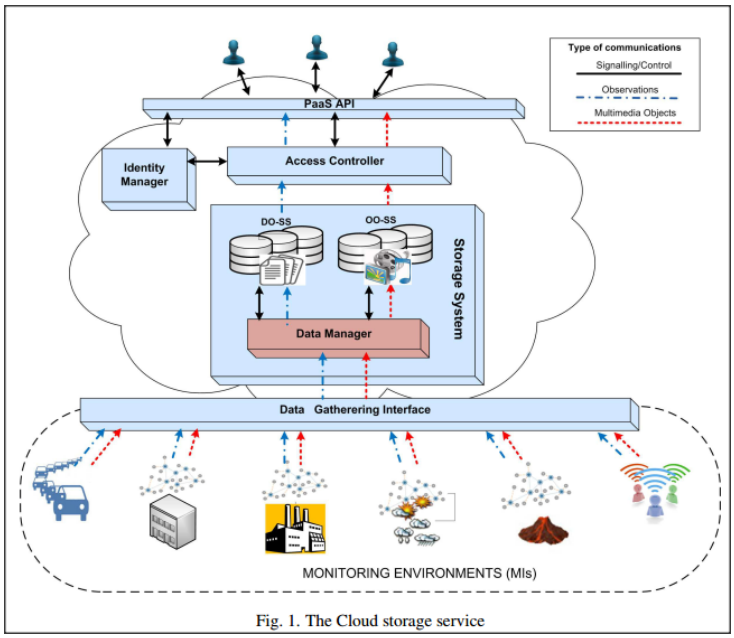
Figura 14 – Arquitetura HiConcept



Em (FAZIO et al., 2015), os autores propõem uma arquitetura para armazenamento em nuvem de dados multimídia de IoT que representa a terceira camada arquitetural para o projeto SIGMA, responsável por agrupar várias redes de diferentes sensores de IoT. A arquitetura de armazenamento proposta se aproveita de requisições REST para obter dados dos sensores e

armazenar no banco de dados MongoDB. Além disso, a arquitetura faz uso da plataforma de armazenamento Swift, que, analogamente ao Amazon S3, fornece várias funcionalidades, como realizar replicação dos dados para evitar ponto único de falha. Apesar de direcionada para dados multimídia e utilizando MongoDB, os autores informam que cada tipo de dado demanda um tipo NoSQL adequado. A Figura 15 demonstra os detalhes da arquitetura proposta para o projeto SIGMA.

Figura 15 – Arquitetura de armazenamento o para projeto SIGMA



Uma outra característica presente na arquitetura é o uso do conjunto de especificações OGC-SWE para tratar os dados obtidos dos sensores. Dois padrões presentes nesta especificação, o *Sensor Observation Service (SOS)* e o *Sensor Alert Service (SAS)*, são utilizados. o SOS é responsável pela requisição, filtros e obtenção de dados de sensores. Já o SAS tem com função realizar a inscrição e publicação de novos sensores. Um outro padrão utilizado é o *SensorML*, que é uma linguagem responsável por prover modelos e esquemas *XML* para descrever dados e processos de sensores. A Figura 16 demonstra a organização destes padrões na arquitetura.

Em (HUO et al., 2016), os autores propõem um *framework* para armazenamento em nuvem de dados de sensores industriais. É proposto um modelo de armazenamento híbrido, onde há a presença de armazenamento relacional e não relacional, sendo cada um destes usado de acordo com a necessidade da aplicação demandante. Diversos tipos de sensores podem ser fonte de dados para a arquitetura, como por exemplo: temperatura, umidade, velocidade do vento, etc., e a arquitetura contempla uma camada de *middleware* que fica responsável por realizar a ponte entre os dados dos sensores e a camada de armazenamento da arquitetura. Os autores sugerem o uso de HBase e Memcached para lidar com dados não estruturados. MySQL e Impala para dados estruturados. A Figura 17 detalha a arquitetura proposta por (HUO et al., 2016).

Em (JIANG et al., 2014), os autores propuseram um *framework* genérico para arma-

Figura 16 – Padrões sensoriais da arquitetura de armazenamento para o projeto SIGMA

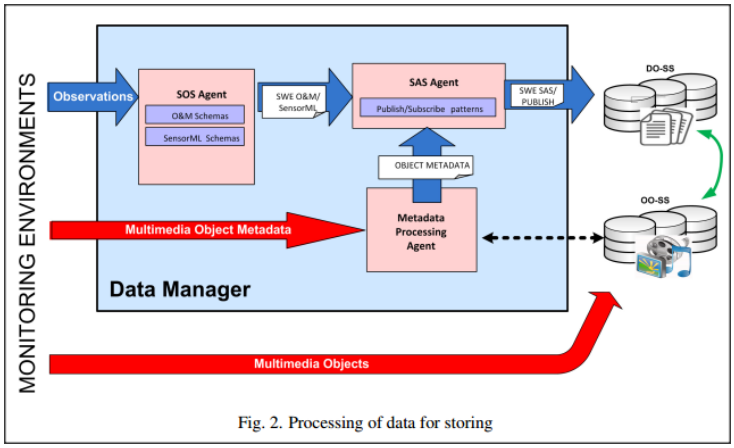
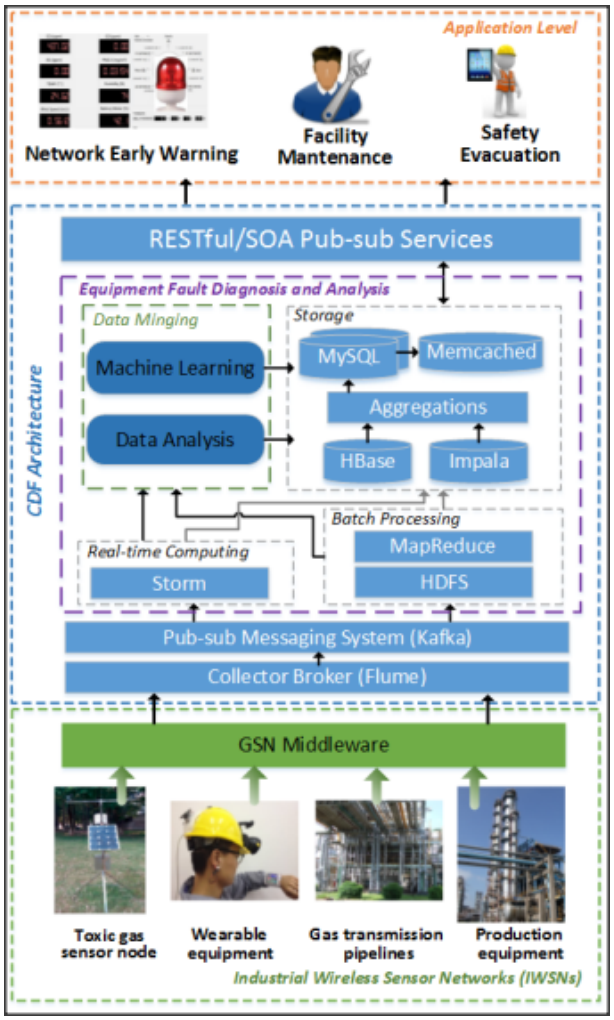


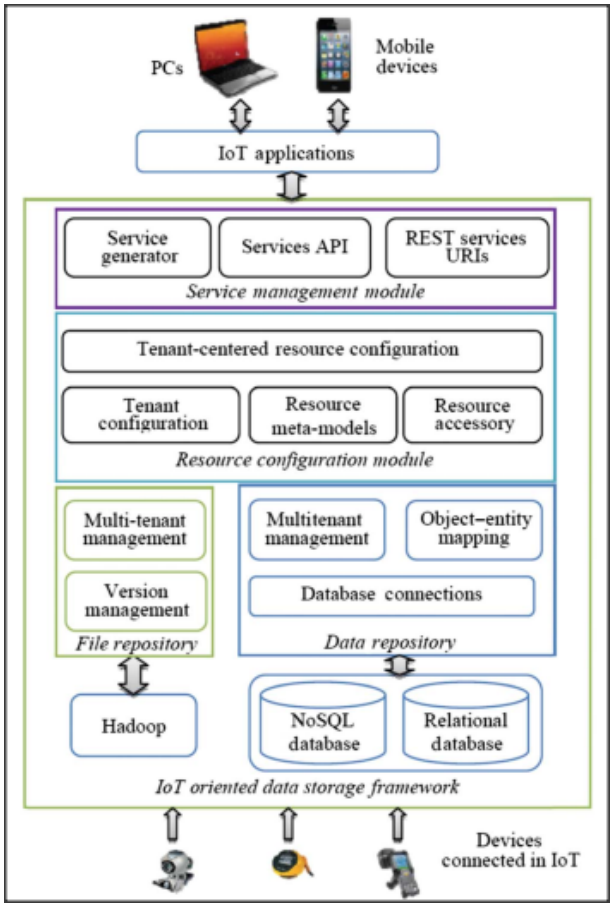
Figura 17 – Arquitetura para armazenamento de dados de sensores industriais



zenamento em nuvem de dados de IoT. O modelo proposto, igualmente ao trabalho de (HUO et al., 2016), também direciona para um modelo híbrido de armazenamento contendo bases relacionais para dados estruturados (MySQL) e bases não relacionais para dados não estruturados (MongoDB). Também de forma similar a trabalho de (FAZIO et al., 2015), faz uso de requisições

REST na arquitetura, só que nesse caso para obter os dados da camada de armazenamento e disponibilizar para as aplicações de IoT. A Figura 18 contém os componentes da arquitetura proposta por (JIANG et al., 2014).

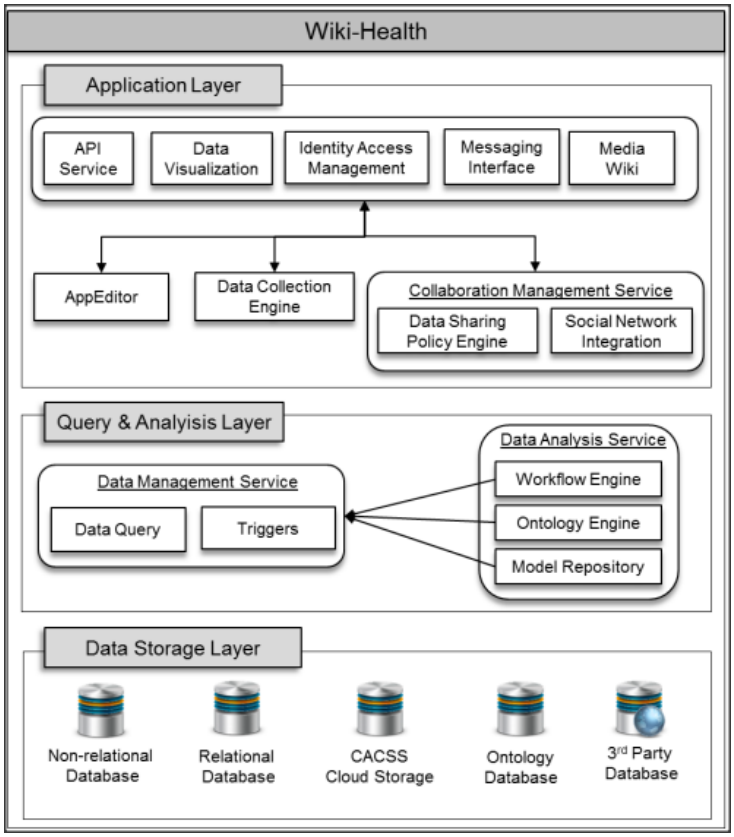
Figura 18 – Arquitetura para armazenamento de dados de sensores de IoT



No modelo exposto, há a presença dos componentes *Service Module* e *Resource Module*, *File Repository* e *Data Repository*. O *Service Module* é responsável por extrair metadados e controlar as requisições REST, realizando o mapeamento das entidade e dados armazenados e gerando os serviços RESTful correspondentes. Já o componente *Resource Module* tem como papel de gerenciar meta-modelos e controlar ajustes de balanceamento de carga. O componente *File Repository* faz uso de do HDFS para armazenar dados não estruturados de forma distribuída. Esse componente pode controlar também o gerenciamento de versões e de camadas. Por fim, o componente *Data Repository* é responsável por controlar o armazenamento híbrido de dados estruturados e não estruturados citados inicialmente.

Em (LI et al., 2014), os autores propuseram uma plataforma para gerenciamento de dados de sensores de saúde, por nome de Wiki-Health onde informações como dados de exames são coletados e armazenados. Igualmente ao trabalhos de (HUO et al., 2016) e (JIANG et al., 2014), um modelo híbrido novamente é proposto, contendo uma camada de armazenamento com bases relacionais (MySQL) não relacionais (HBase). A Figura 19 contém os componentes da arquitetura Wiki-Health.

Figura 19 – Arquitetura Wiki-Health para armazenamento de dados de sensores de saúde



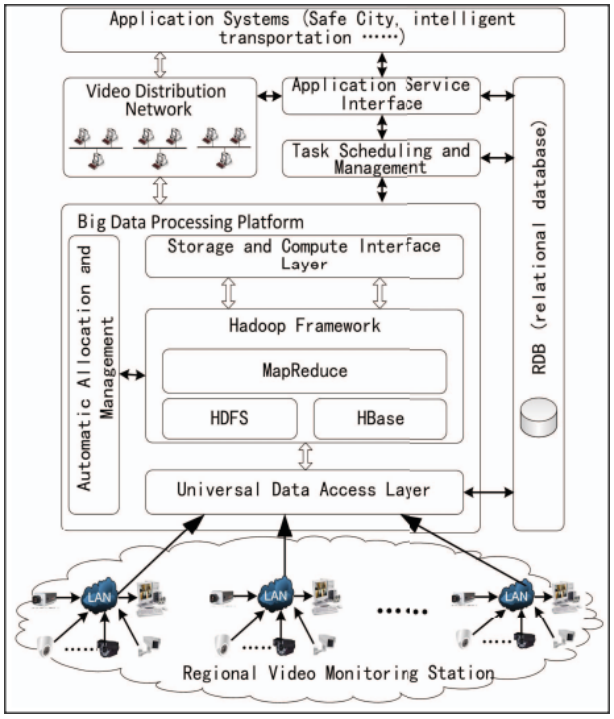
Contudo, diferente dos trabalhos já explicados, neste trabalho os autores adicionam mais 3 abstrações possíveis para armazenamento dos dados: uma chamada *CACSS Cloud Storage*, uma base para ontologias e uma base denominada de terceira parte. O CACSS é um sistema em nuvem usado para guardar dados não estruturados adicionais. A base de ontologia é utilizada para análises de Big Data e a terceira parte para armazenar dados de outras categorias.

Em (LIN et al., 2014), os autores propõem um sistema de monitoramento de mídias de vídeo. Igualmente aos trabalhos anteriores, propõe um modelo de armazenamento híbrido, relacional e não relacional, onde o sistema possui dois objetivos: lidar com a quantidade massiva de dados não estruturados, e realizar o tratamento e análise dos dados no contexto de Big Data. A Figura 20 contém os componentes da arquitetura por (LIN et al., 2014).

É possível notar que o sistema se utiliza do *framework* hadoop e das bases de armazenamento não relacional HDFS e HBase. A camada de *Universal data Access Layer* é responsável por coletar vários de tipos de arquivos de vídeos, os quais são armazenados no HDFS ou no HBase após a correto tratamento dos mesmos.

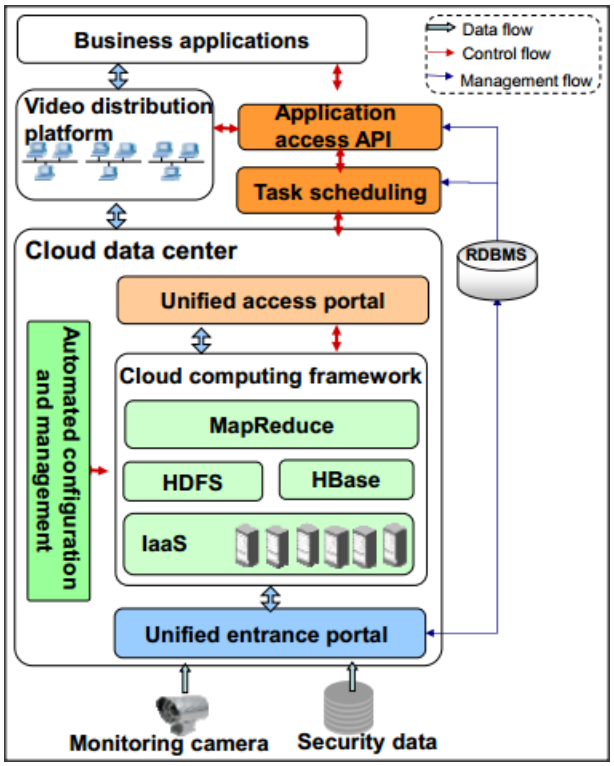
Em (LIU; WU; LIAN, 2015), os autores criam uma arquitetura para armazenamento de dados de vídeo, que, com estrutura muito próxima ao trabalho de (LIN et al., 2014), onde existe uma camada chamada *Unified Entrance Portal*, que serve como ponto de entrada para os dados de vídeos. Também utiliza uma estrutura em nuvem e conta com armazenamento híbrido

Figura 20 – Arquitetura para armazenamento de arquivos de vídeo de IoT



contendo armazenamento relacional e não relacional (HDFS e HBase). A Figura 21 contém os componentes da arquitetura de (LIU; WU; LIAN, 2015).

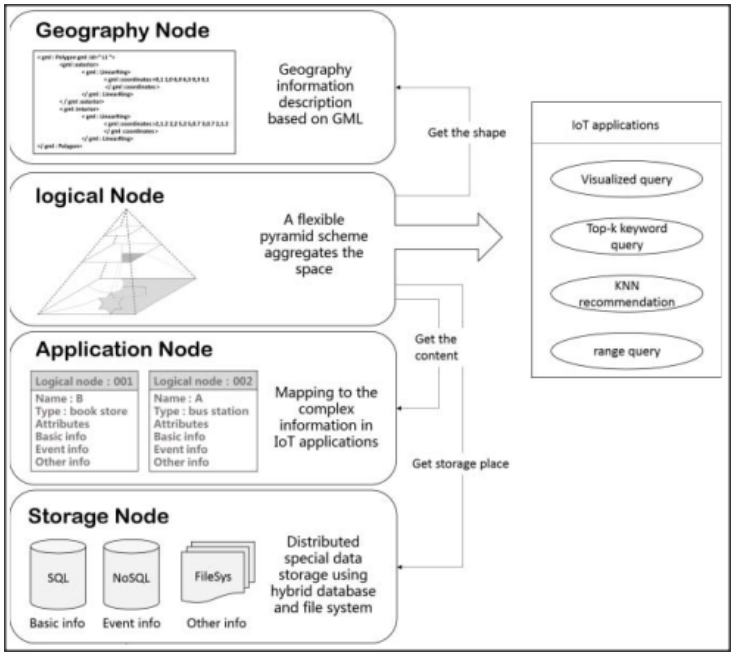
Figura 21 – Arquitetura para armazenamento de arquivos de vídeo de IoT de (LIU; WU; LIAN, 2015)



Em (LUAN et al., 2015), os autores propuseram uma arquitetura para armazenamento de

dados espaciais (incluindo posicionais) contendo 4 camadas: geográfica, lógica, de aplicação e de armazenamento. A camada geográfica serve para descrever o local e o formato de um local. A camada lógica representa o ponto central da arquitetura. Um dos seus papéis é obter informações posicionais de um objeto. Outra função dessa camada é gerenciar de forma eficiente o processo e a obtenção de dados espaciais. A camada de aplicação tem como papel descrever a estrutura de objetos, como atributos, herança, e também informações dinâmicas do estado de um objeto. A camada de armazenamento tem a importante função de direcionar o local correto de armazenamento de um dado a depender do seu tipo, considerando, mais uma vez, um modelo híbrido de armazenamento, onde, a arquitetura proposta exemplifica o uso de MongoDB. A Figura 22 contém os componentes da arquitetura de (LUAN et al., 2015).

Figura 22 – Arquitetura para armazenamento de dados espaciais



Em (MA et al., 2012), os autores propõem uma arquitetura para lidar com dados de informações de tráfego veicular, como a velocidade, posição, direção e destino de veículos. Nesse trabalho também é proposto um modelo híbrido onde informações sociais são armazenadas utilizando o modelo relacional e as informações referentes ao tráfego, que são não estruturadas, utilizando o modelo não relacional; onde é importante frisar que dois desses dados chave são a posição e a velocidade do veículo, informações colhidas através de dispositivos do tipo *GPS receiver*. No modelo os autores propõe um uso de um estrutura em nuvem contendo 3 camadas: IaaS, PaaS e SaaS. A parte de armazenamento em nuvem fica na camada de IaaS, onde várias tecnologias são propostas, entre elas o uso do banco de dados HBase e também utilizando MapReduce. A Figura 23 contém os componentes da arquitetura de (MA et al., 2012).

Em (MA; MOTTA; LIU, 2017), analogamente ao trabalho de (MA et al., 2012), também propõe uma arquitetura, por nome de MOBANA, para armazenamento de informações de tráfego veicular. Contudo, algumas diferenças estão presentes. Em (MA; MOTTA; LIU, 2017), é

Figura 23 – Arquitetura para armazenamento de dados de tráfego veicular

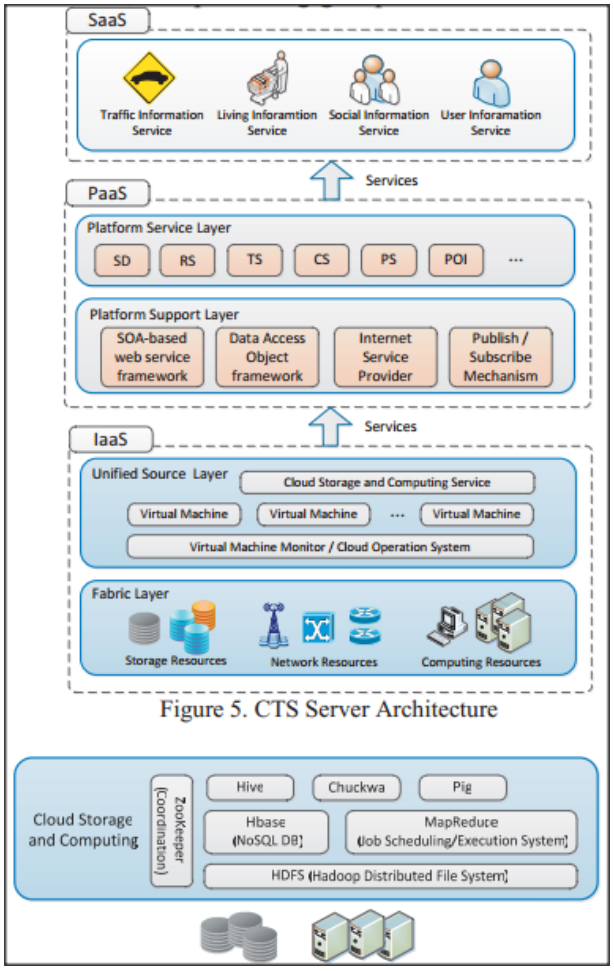


Figure 5. CTS Server Architecture

proposto um modelo onde se faz uso de requisições RESTfull para obtenção e encaminhamento dos dados na plataforma. Com uso de RESTfull, o modelo faz uso do banco de dados MongoDB, ao invés do HBase. O MongoDB tem um importante papel no *framework* de servir como fonte para estudos analíticos no contexto de Big Data. Contudo, o trabalho também faz uso de um modelo arquitetural híbrido, onde é utilizando também o banco de dados relacional PostgreSQL. A Figura 24 contém os componentes da arquitetura MOBANA.

Em (RONKAINEN; IIVARI, 2015), os autores definem uma arquitetura genérica de alto nível para lidar com o armazenamento de dados de sensores. Devido as características de prover alto desempenho e escalabilidade, é indicado o uso de bases de dados de família de colunas. Nesse âmbito, entre as bases citadas, é aconselhado o uso de HBase e Cassandra. Na arquitetura definida, os autores também indicam o uso de requisições HTTP com REST para obtenção dos dados dos sensores. Além disso, algumas ferramentas para análise de dados são indicadas. A Figura 25 contém os componentes da arquitetura de (RONKAINEN; IIVARI, 2015).

Em (YANG; WANG; ZHAO, 2016), os autores propõem um esquema para identificação de sensores utilizando sua URI. O modelo é chamado de URIoT, utiliza requisições *JSON* para troca de dados onde além da URI com a localização do sensor na estrututura, ainda vem com os

Figura 24 – Arquitetura para armazenamento de dados de tráfego veicular MOBANA

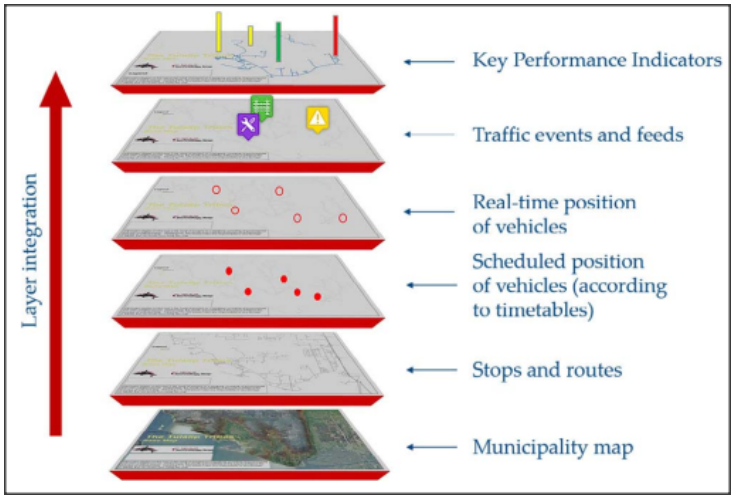
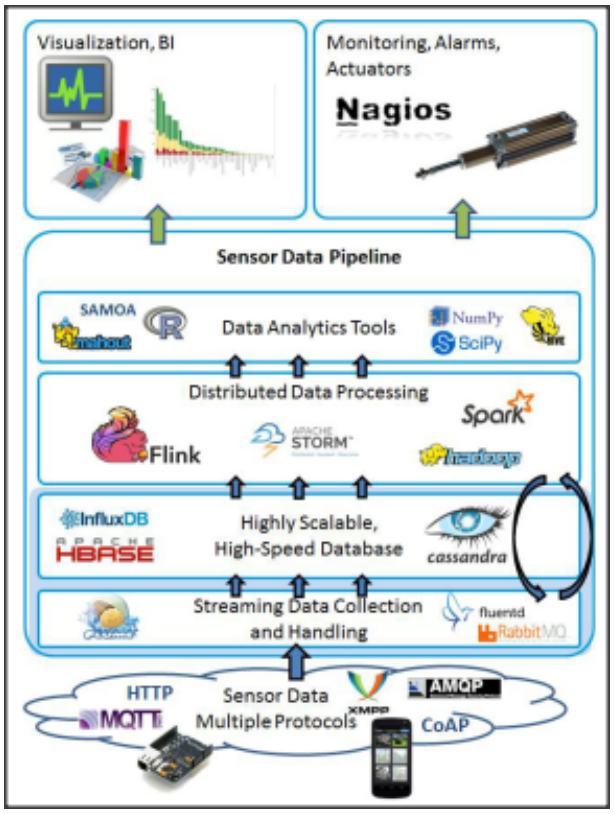


Figura 25 – Arquitetura para armazenamento de dados de sensores utilizando família de colunas



demais dados referentes ao objeto. A Figura 26 mostra a estrutura dos objetos de URIoT segundo sua URI e a Figura 27 um exemplo de requisição *JSON* de um objeto do modelo.

Devido à baixa capacidade de armazenamento e memória dos dispositivos, a comunicação entre estes e o servidor de registro segue um modelo leve publicação e inscrição, utilizando o protocolo MQTT, que garante transmissão de dados de forma segura e confiável. Os autores utilizam uma biblioteca chamada de Paho Python que suporta o uso de MQTT. O esquema criado utiliza a base de dados NoSQL Redis, que utiliza a própria memória e cache dos dispositivos

Figura 26 – Estrutura dos objetos em URIoT

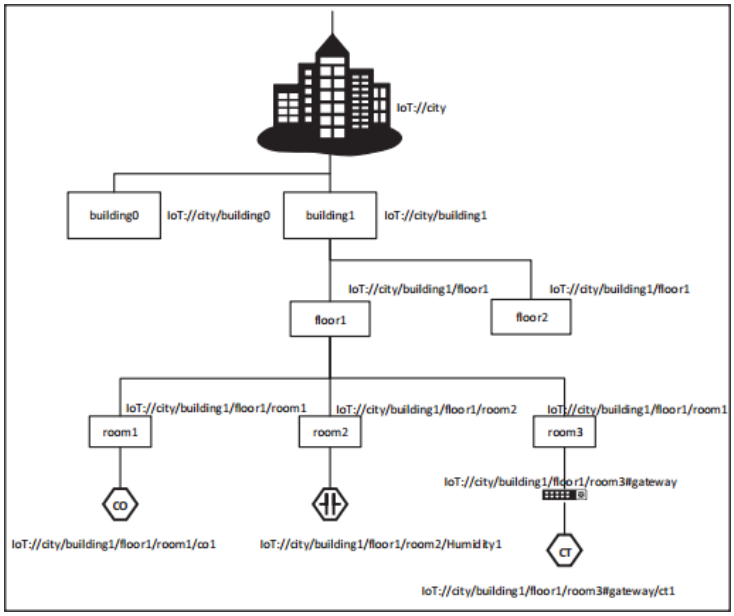
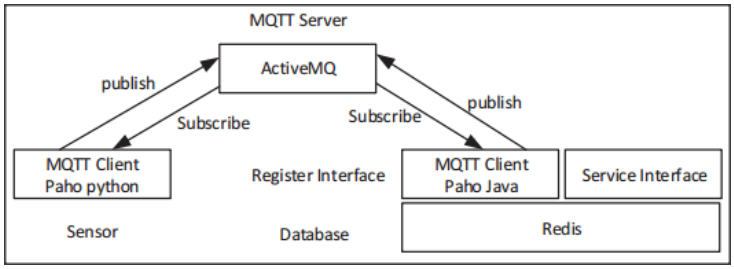


Figura 27 – Exemplo de requisição *JSON* do URIoT


```
{
  “id”: “IoT://city/building1/floor1/room2/T1”,
  “name”: “T1”,
  “type”: “Temperature”
  “frequent”: “0.5Hz”,
  “dataType”: “float”,
    “range”: [
       -20,
       100
  ],
  “accuracy”: “0.1”,
  “unit”, “℃”,
  “perception object”: “IoT://city/building1/floor1/room2”
}
```

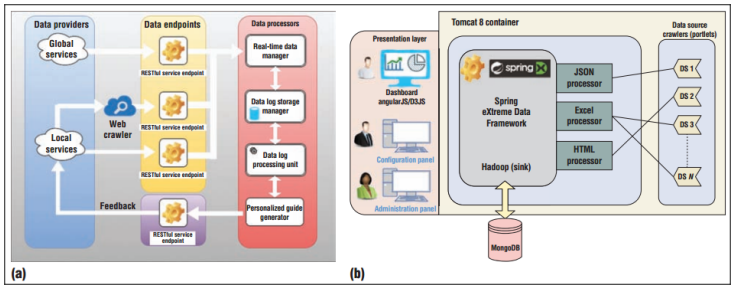

para armazenar dados, para realizar o armazenamento dos dados e informações de registros dos dispositivos. A Figura 28 mostra como ocorre a troca de informações entre os dispositivos e o servidor em URIoT.

Figura 28 – Troca de dados entre servidor de inscrição e dispositivos na URIoT



Em (ZDRAVESKI et al., 2017), os autores definem uma arquitetura de alto nível, baseada na ISO 37120, para aquisição e processamento de dados de sensores em cidades inteligentes. Chamada de ISO4City, a arquitetura contém a definição de um *framework* para obtenção e tratamento dos dados que contém rastreadores de dispositivos para inclusão e inscrição na arquitetura, utilizado o serviços RESTful e fazendo uso do banco de dados MongoDB, tratado no trabalho como ideal para lidar com dados de não estruturados de sensores. A Figura 29 representa o *framework* de gerenciamento dos dados da arquitetura ISO4City.

Figura 29 – *Framework* de gerência de dados da Arquitetura ISO4City (a) e sua implementação (b)



Em (AYDIN; HALLAC; KARAKUS, 2015), os autores definem um arquitetura para armazenamento de dados posicionais (uso de GPS) para sensores de IoT. A arquitetura utiliza o padrão *JSON* para direcionar os dados obtidos dos sensores para o armazenamento nas bases de dados em MongoDB, além de fazer de HDFS e Hadoop para tratar os dados e tornar possível a análise dos mesmo em análises de Big Data. Os autores justificam também que o uso do MongoDB devido a sua capacidade de realizar fragmentação de dados, recuperação de falhas automática e alto desempenho de escrita de dados. A Figura 30 representa os componentes da arquitetura de (AYDIN; HALLAC; KARAKUS, 2015).

Em (FAZIO; PULIAFITO; VILLARI, 2014), os autores propuseram uma arquitetura de alto nível, chamada IoT4S, para lidar com o gerenciamento de dados de sensores. A arquitetura está conceitualmente dividida em 4 camadas: *Interface, SOS Agent, Sensor Manager e Sensing Infrastructure*. A camada de *Interface* é responsável por prover serviços para que as aplicações possam obter os dados tratados e armazenados dos sensores utilizando o padrão REST. A camada *SOS Agent* tem como função principal por gerenciar um modelo híbrido de armazenamento de dados de sensores utilizando bancos relacionais para comunicação interna e bases não relacionais para comunicação externa (indicado o uso de Cassandra). A camada *Sensor Manager* tem como responsabilidades detectar dispositivos e coletar dados dos sensores. Por fim, a camada de *Sensing Infrastructure* é a que comporta todos os tipos de dispositivos (sensores) que alimentam a arquitetura com dados. A Figura 31 representa os componentes da arquitetura IoT4S.

Em (CAI et al., 2016), os autores propõem uma arquitetura para gerenciar dados de sensores composta de 3 camadas e uma *interface*: *IoT Interface, O2O Application Layer, Multi-layer IoT Data Management Layer, Physical Storage Layer*. A *IoT interface* representa o

Figura 30 – Arquitetura para armazenamento de dados posicionais com MongoDB

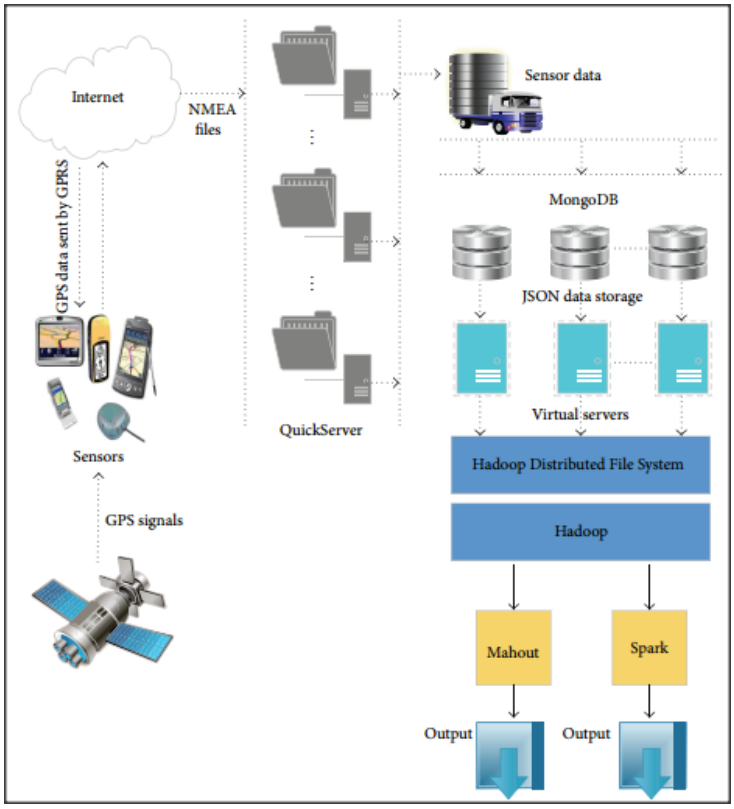
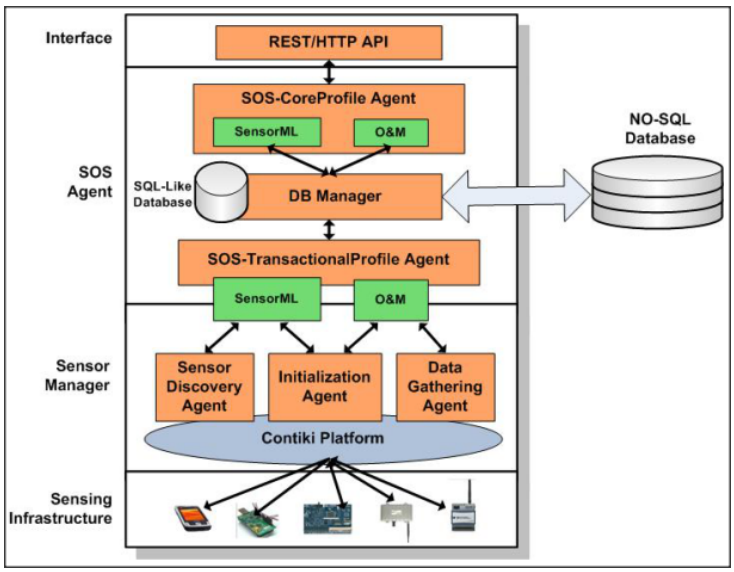


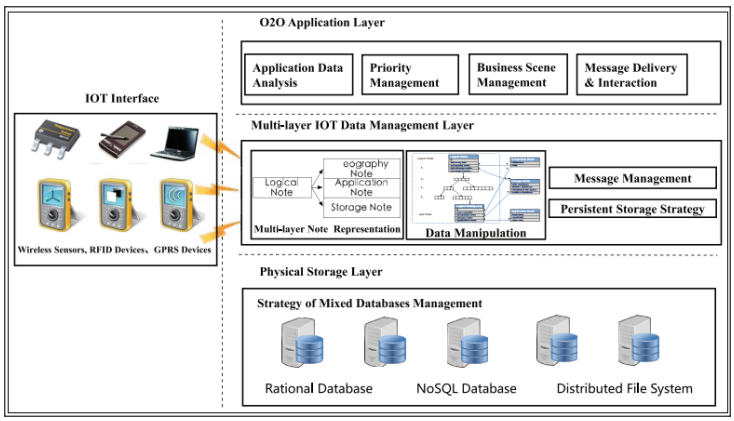
Figura 31 – Arquitetura para armazenamento de IoT4S



componente que comporta os dispositivos de IoT que alimentam a arquitetura com dados, como por exemplo, sensores sem fio, dispositivos RFID e dispositivos GPRS. O componente *Multi-layer IoT Data Management Layer* é a camada responsável por gerenciar a recepção e correto tratamento e direcionamento dos dados dos sensores. Ou seja, é nessa camada que tem ligação direta com o componente *IoT interface*. Os dados são corretamente direcionados para a camada *Physical Storage Layer*, que lida com a estratégia de armazenamento híbrido dos

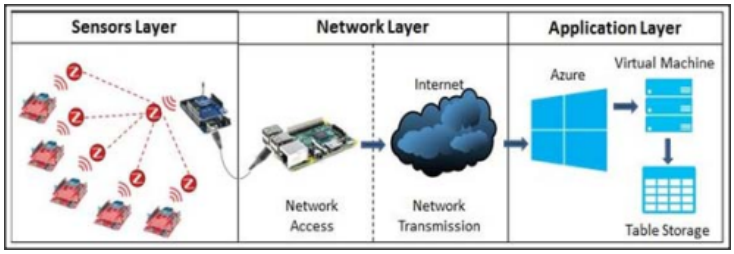
dados dos sensores a depender do seus requisitos. Com isso, a arquitetura proposta também provê um modelo híbrido de armazenamento relacional e não relacional, onde é indicado o uso de MongoDB para armazenamento não relacional. Já o componente *O2O Application Layer* representa a camada das aplicações que farão uso dos dados dos sensores já tratados e armazenados na camada *Physical Storage Layer*. A Figura 32 representa os componentes da arquitetura de (CAI et al., 2016).

Figura 32 – Arquitetura para armazenamento híbrido de dados de sensores



Em (VANELLI et al., 2017), os autores propõem uma infraestrutura de armazenamento de dados de IoT em nuvem composta de 3 camadas: camada de sensores, camada de rede e camada de aplicação. A camada de sensores tem por responsabilidade servir para coleta e transmissão de dados, além de servir de ponte para a camada de rede. Essa camada é formada por uma rede Zigbee, que tem como característica ser de baixo custo e consumo energético. Nesta camada, 3 tipos de dispositivos estão presentes: um sensor de temperatura, o DHT11 (CITAR), um resistor LDR, onde a resistência varia conforme a intensidade de luz; e um sensor passivo infravermelho. Todos esses componentes estão interligados numa *protoboard* Arduino. A segunda camada é a de rede. Nela, existe um *gateway* IoT que transforma dados de sensores em requisições para a camada de aplicação. Já a camada de aplicação tem APIs que captura os dados do *gateway* de IoT e possibilitam seu armazenamento em uma base não relacional (Azure DB) em nuvem. A Figura 33 representa os componentes da arquitetura de (VANELLI et al., 2017).

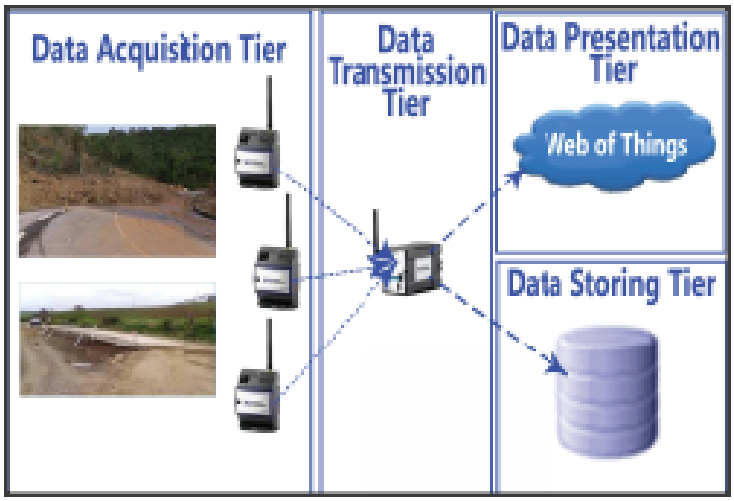
Figura 33 – Arquitetura para armazenamento de dados de sensores de (VANELLI et al., 2017)



Em (KEBAILI et al., 2016), os autores definem uma arquitetura por nome de LEWS, para monitorar o nível de deslizamento de terra em uma área de risco na Tunísia. A arquitetura é

dividida em 4 camadas: Camada de aquisição de dados, camada de transmissão de dados, camada de armazenamento e camada de apresentação dos dados. Na camada de aquisição dos dados estão a rede de sensores de fio e sensores de precipitação, umidade e acelerômetro. A camada de transmissão é a responsável por obter os dados. A camada de armazenamento dos dados é a que gerencia os dados coletados dos sensores utilizando o banco de dados de documentos MongoDB; e a camada de apresentação é a responsável por fornecer os dados para visualização por usuários e outras aplicações. A Figura 34 demonstra a estrutura e camadas da arquitetura LEWS.

Figura 34 – Camadas da arquitetura LEWS



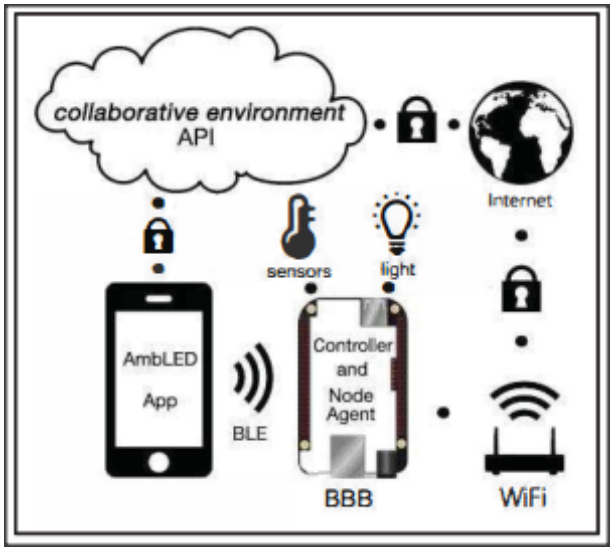
4.2 Contextos Experimentais Relacionados

Em (FRANCESCO et al., 2012), os autores realizaram experimentos com dados escalares e multimídia de sensores de IoT, através de 50 repetições, com foco na obteção do tempo médio de requisição e resposta de armazenamento dos dados na base de dados CouchDB. Para coletar as informações, os autores utilizam um *framework* proposto por (FRANCESCO et al., 2011), onde há a existência de uma *protoboard* do tipo BeagleBone com um sensor multimídia integrado com uma câmera que tira várias *snapshots* do ambiente numa casa de cuidados de pessoas idosas e disponibiliza as imagens para posterior utilização. Além da câmera, os autores informam que outros tipos de sensores (temperatura, umidade, pressão) podem ser adicionados. Com isso, o a avaliação do tempo médio de armazenamento no CouchDB pode ser realizado para dados escalares (numéricos), posicionais (textuais ou numérico) e dados multimídia (binários).

Em (CUNHA; FUKS, 2015), os autores criaram um protótipo, por de AmbLEDs, onde realizaram um experimento com uma *protoboard* do tipo BeagleBone Black (BBB) para auxiliar na administração de medicações em pacientes de um hospital. Com a *protoboard* e sensores interligados, a ideia é controlar um lâmpada ligada a gavetas, onde, caso o medicamento retirado das gavetas seja o correto e o horário de administrar o medicamento seja também o correto, uma luz verde acenderá na lâmpada e um som será emitido. Caso o medicamento ou o horário

não seja o correto, uma luz vermelha e um som diferente do anterior será emitido. Os dados referentes às medicações e horários são armazenados em uma base MongoDB. Além disso, para troca de dados entre a *protoboard* e as aplicações, foi utilizado o padrão BLE. A Figura 35 representa os componentes da arquitetura.

Figura 35 – Arquitetura do protótipo AmbLEDs



Em (VANELLI et al., 2017), os autores realizaram experimentos com sensores de temperatura utilizando uma *protoboard Arduino* numa rede *Zigbee*. O objetivo é avaliar o tempo médio de armazenamento de dados dos sensores no banco de dados de família de colunas Azure DB, utilizando um ambiente em nuvem. Para possibilitar a captura de dados dos sensores na *protoboard* e seu correto armazenamento na base de dados em nuvem, foram utilizadas APIs com a linguagem de programação Python.

Em (HASHI et al., 2015), os autores criam um esquema para armazenamento de dados de sensores utilizando o banco de dados CouchDB e o banco Neo4j. Para avaliar o modelo, os autores realizaram experimentos para avaliar o tempo de consulta de vários conjuntos de dados coletados de sensores. Dados dos sensores eram coletados a cada 5 minutos, gerando um total de 288 registros por dia. Foram aferidos os tempos de consultas de vários conjuntos de dados nas faixas de 100, 1000, 10000 e 100000 dias. Além disso, também foram aferidos dados em sensores de 100, 1000, 10000 e 100000 sensores. Para a realização dos testes, os autores utilizaram uma rede ligada via *switching hub* de 100Mbps e utilizaram a infraestrutura presente na Figura 36

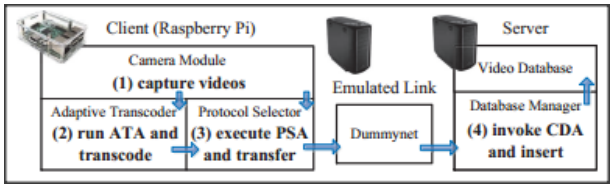
Em (HONG et al., 2016), os autores fazem experimentos de inserção, deleção e consultas de dados multimídia (vídeos), capturados utilizando uma *protoboard* do tipo Raspberry Pi, utilizando uma câmera. No experimento, os vídeos capturados são armazenados em um servidor em nuvem na base de dados MongoDB. O foco no experimento foi a avaliação de desempenho dos tempos de inserção deleção e consultas no MongoDB, MySQL e PostGIS. Como resultados,

Figura 36 – Infraestrutura usada nos experimentos de consulta de (HASHI et al., 2015)

	Server	Client
CPU	Core i7-3.9Ghz, 4Core/8Thread	Core i3-1.70GHz, 2Core/4thread
Memory	8GB	8GB
Disk	256GB SDD	256GB SDD
OS	CentOS Linux7.0.1406 64bit	Windows7 Pro. 64bit

com o MongoDB a inserção e a deleção foram rápidas, mas as consultas foram lentas. A Figura 37

Figura 37 – Infraestrutura usada nos experimentos de (HONG et al., 2016)



Em (KANG et al., 2016), os autores desenvolveram um esquema de armazenamento de dados de simulações de partes de automóveis utilizando dados de sensores, os quais são validados em experimentos em relação a tempo de consultas usando o banco de dados MongoDB. São realizados experimentos comparativos em relação ao desempenho de consulta desses dados entre o MongoDB e o MySQL, onde, como resultado, o MongoDB tem melhor resultado se o desempenho da consulta for o único requisito. Foram definidas algumas categorias de testes: consulta dos dados em um periodo específico de tempo e em um lugar específico (Q1), consulta dos dados em um periodo específico de tempo (Q2), consulta dos dados em uma hora específica utilizando um ID (Q3). Para cada condição destes, consultas específicas no MongoDB foram criadas. A Figura 38 mostra as condições e consultas.

Figura 38 – Lista de testes e consultas com MongoDB em (KANG et al., 2016)

Query	Description	Query parameters	Query Statement
Q1	Query for event that occurred in a specific period and a specific place	readPoint, eventTime	db.collection.find({ readPoint: "readPoint x", eventTime:{ $gte:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'"), $lt:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'") });
Q2	Query for event related to a target product during a specific period	eventTime, epcList	db.collection.find({ epcList: "epc x", eventTime:{ $gte:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'"), $lt:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'") });
Q3	Query for object ID contained in a specific case at specific time	eventType, eventTime, parentID	db.collection.find({ parentID: "epc x", eventType: "AggreationEvent", eventTime:{ $gte:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'"), $lt:ISODate("yyyy-MM-dd'T'HH:mm:ss'Z'") });

Em (LU; FANG; LIU, 2016), os autores propuseram um *framework*, por nome de DeCloud-RealBase, onde foram realizados 3 experimentos para sua validação utilizando as bases HBase e MySQL. Os experimentos de consulta foram realizados com um total de 280 milhões

de registros de dados de sensores coletados de veículos. A Figura 39 mostra o ambiente utilizado nos experimentos e a função de cada ferramenta.

Figura 39 – Ferramentas utilizadas no *framework* DeCloud-RealBase

Equipment	Environment	Function
LoadRunner Server (4)	Dual core 3.0GHz CPU、4 GB Memory	Simulate the sending of sensor data
DeCloud-RealBase Server(1)	2×4 core 2.4 GHz CPU、16 GB Memory	Enact the unified storage and optimization framework *DeCloud-RealBase*
MySQL Database Server (4)	2×4 core 2.4 GHz CPU、8GB Memory	Persistent data storage, one for Mycat1.3, others for Mysql5.1
HBase Database Server (4)	2×4 core 2.4 GHz CPU、8GB Memory	Long-term data storage, one acts as master server and others act as slave servers.

No primeiro experimento, usando o servidor de carga (*LoadRunner Server*), foi simulado o envio de dados em diferentes quantidades: 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000 e 40000 por segundo, onde os dados possuem tamanho de 100 bytes. O segundo experimento foi de consultas de 5 conjuntos de dados. O último experimento foi também de consulta, mas desta vez avaliando os tempos com e sem o uso de *cache*.

Em (MOAWAD et al., 2015), os autores realizaram experimentos de escrita e leitura, nos bancos LevelDB e MongoDB, em 7 conjuntos de dados de sensores: Dados numéricos, lineares, sensores de temperatura, luminosidade, eletricidade, arquivos de música e dados aleatórios. A Tabela 40 mostra o ambiente utilizado nos experimentos e a função de cada ferramenta.

Figura 40 – Tabela de tipos de dados usados em (MOAWAD et al., 2015)

Database	Sensor
DS1: Constant	c=42
DS2: Linear function	y=5x
DS3: Temperature	DHT11 (0 50'C +/- 2'C)
DS4: Luminosity	SEN-09088 (10 lux precision)
DS5: Electricity load	from Creos SmartMeters data
DS6: Music file	2 minutes samples from wav file
DS7: Pure random	in [0;100] from random.org

Em (RADU et al., 2016), os autores realizam experimentos de escrita e deleção de dados utilizando as bases MongoDB, CouchDB e Cassandra, de dados de sensores de um *framework* em nuvem, chamado SmartFarm, para gerenciar fazendas. O principal resultado obtido é que o MongoDB obteve o pior desempenho no caso de deleção de dados em relação ao CouchDB e ao Cassandra, o que, segundo os autores, ocorre devido o MongoDB efetivamente deletar o dado fisicamente, enquanto que no CouchDB e no Cassandra o dado apenas é marcado como excluído.

Em (WANG et al., 2016), os autores realizam experimentos de escrita e leitura de dados de dois tipos de sensores de temperatura, ligados em uma rede Zigbee, utilizando a biblioteca e uma sala de manutenção de uma universidade e armazenando os dados nos bancos HBase e MySQL. O resultado dos testes é que o HBase, utilizando *MapReduce*, obteve tempos menores nas operações. A Tabelas 41 e 42 mostram as ferramentas utilizadas no experimento.

Figura 41 – *Hardware* do cliente e do servidor em nuvem em (WANG et al., 2016)

Hardware and Software for all service		
Web Server *1	CPU	Intel(R) Core(TM)2 CPU6420@ 2.13GHz
	RAM	2GiB DIMM SDRAM Synchronous*2
	HDD	250GB Hitachi HDT72502*2
	OS	Linux Ubuntu 12.04.5 LTS
Cloud Server *1	vCPU	4 cores
	RAM	8GB
	HDD	100GB
	OS	Linux Ubuntu 12.04 LTS

Figura 42 – Sensores de temperatura usados em (WANG et al., 2016)

Hardware for Sensing service		
Sensor no.	Sensor Type	Specification
Sensor 1	Temperature and humidity	Series WHT
	Formaldehyde	CTX 300
	VOC	OLCT 100 XP
	Carbon monoxide	OLCT 20 D
Sensor 2	CO2, Temperature, Humidity	ZGw08VRC

Em (ZHAI et al., 2016), os autores definem um *framework* para capturar dados de sensores de temperatura utilizando a plataforma MoteStack. Em posse dela, e utilizando a ferramenta de avaliação de desempenho JMeter, experimentos de desempenho e concorrência foram realizados utilizando taxas de transmissão de mensagens armazenadas no banco de dados MongoDB. Os testes foram realizados com o auxílio de um *middleware* para obtenção dos dados na plataforma MoteStack e armazenamento no MongoDB. Como resultado principal, foi atestado que 99% das requisições ocorridas são completadas em menos de 1 milisegundo.

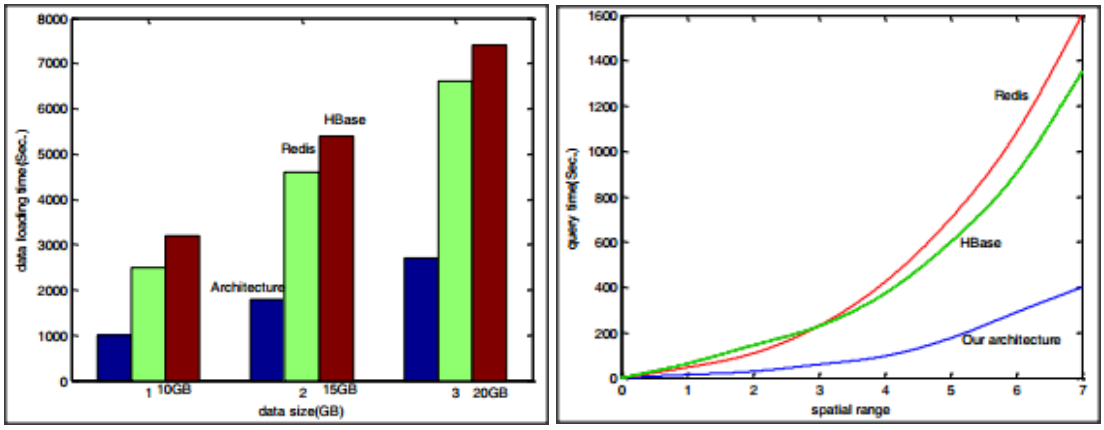
Em (ZHANG et al., 2013), os autores realizam experimentos para avaliar o tempo de carregamento e e o tempo de consultas de dados de sensores de temperatura, umidade e concentração de dióxido de carbono. Para o experimento foram utilizadas as bases Redis, HBase e Oracle e foram feitas comparações com o *framework* desenvolvido. Segundo os autores, os o tempos nos experimentos foram menores na arquitetura proposta. A Figura 43a mostra o desempenho no carregamento dos dados e a Figura 43b mostra o desempenho nas consultas.

Em (PHAN et al., 2014), são realizados testes de dois tipos de dados de IoT, dados escalares e dados multimídia, utilizando exemplos de bases de dados NoSQL e também relacional. O foco do trabalho foi avaliar o desempenho de inserção e recuperação de dados nas bases testadas de forma a conferir qual possui o melhor custo/benefício. Foram realizados testes com dois tipos de bancos de dados NoSQL e com um tipo de base relacional. Entre os tipos NoSQL, foram realizados testes com bancos de dados de documentos, utilizando os *clients* CouchDB e MongoDB; utilizou-se também um exemplo de banco de dados do tipo chave-valor, com o *client* Redis. Já com o modelo relacional, utilizou-se o *client* MySQL.

O experimento inicial utilizou dados escalares de simulações em IoT. Os bancos de dados MySQL, CouchDB, MongoDB e Redis foram avaliados. Usaram-se conjuntos com apenas uma coleção de dados e outros com várias coleções de dados. Sobre os resultados, o Redis obteve menor tempo tanto na escrita como na leitura de dados. Contudo, vale salientar que o Redis é

Figura 43 – Desempenho nas escritas e consultas em (ZHANG et al., 2013)

(a) Desempenho nas escritas (b) Desempenho nas consultas



dependente do tamanho da memória do cliente e não realiza fragmentação de dados naturalmente. Testes adicionais são necessários avaliar melhor o potencial do Redis. O MongoDB, quando usando índices, obteve menor tempo de escrita e leitura, já que possui fragmentação nativa e uso de índices. Os autores destacam que aumentar o número de índices ocasiona maior latência na escrita. O experimento final consistiu em um comparativo entre o MongoDB e o MySQL para dados multimídia. O resultado final foi inconclusivo. Com isso, mais testes para dados multimídia são necessários.

Alguns pontos sobre esse trabalho pode ser destacados: Entre os tipos NoSQL existentes, não foram testadas bases do tipo famílias de colunas, como os *clients* Cassandra e HBase; e nem bases orientadas a grafos, como o *client* Neo4J, o que não dá uma perspectiva mais amadurecida de qual base de dados NoSQL realmente possui menor tempo de escrita ou leitura para dados escalares e multimídia. Um outro ponto a ser considerado é em relação ao experimento de dados multimídia, que, além de escolher de forma arbitrária o MongoDB, talvez tenha havido inconclusão devido ao uso de apenas 10 casos no experimento, o que pode ser considerado um número pequeno de casos, quando comparado à base relacional MySQL. Outro detalhe é não ter deixado de forma clara como o *testbed* e a massa de dados dos experimentos foram configurados.

Em (Van Der Veen et al., 2012), os autores realizam experimentos de inserção e leitura de dados escalares de sensores, utilizando máquinas físicas e virtualização, em uma base SQL, com o *client* Postgres, e com bases NoSQL Cassandra e MongoDB. Os autores isolam uma rede entre as máquinas para tentar atenuar a interferência externa e fazem testes de desempenho tanto nos bancos das máquinas físicas, como nas virtualizadas. Os principais resultados obtidos foram que a base Cassandra, do tipo família de colunas, é a melhor escolha para aplicações críticas de sensores, onde a virtualização influencia diretamente no desempenho das aplicações; MongoDB é o melhor tipo para aplicações onde os dados de sensores não são críticos, especialmente quando o desempenho na escrita for muito importante; PostgreSQL é a melhor escolha quando consultas

complexas são necessárias e desempenho em leitura é um requisito essencial, além de mostrar que a gravação de poucos dados é lenta, mas que é positivamente afetada pela virtualização.

Em (KEBAILI et al., 2016), os autores realizaram um experimento para prova de conceito da arquitetura LEWS para monitorar o nível de deslizamento de terra em uma área de risco na Tunísia. Para realizar o experimento, os autores utilizaram uma rede estática com topologia em árvore, 5 *waspmotes* e 3 tipos de sensores: precipitação, umidade e um sensor do tipo acelerômetro para monitorar a movimentação do solo, onde foram divididos 5 níveis para o estado do solo: normal, molhado, aumento da umidade, umidade crítica, perigo de deslizamento. Em relação a implementação, foram desenvolvidos duas aplicações: Uma em C++ para obter os dados dos sensores, e outra em Python para pegar esses dados e gravar os mesmos em uma base em MongoDB. A Figura 44 exemplifica dados em *JSON* dos sensores do modelo.

Figura 44 – Exemplo dados coletados da arquitetura LEWS

{ 'id':'5763e19507a38f700800004b',
'mac':'0013a2004070e9af',
'date':'17-06-2016 13:40:05',
'temperature':'27', 'luminosity':'0', 'pressure':'108', 'watermark':'51', 'battery':'28', 'x_acc':'-125', 'y_acc':'92',
'z_acc':'999' }

Em (LE et al., 2014), os autores realizaram uma série de experimentos para avaliar o desempenho de um sistema de informações de códigos de produto (EPCIS) com as bases Cassandra e MySQL. Foram avaliadas 4 categorias de testes entre as duas bases: cliente único com múltiplas inserções, cliente único com múltiplas consultas, múltiplos clientes com múltiplas inserções e múltiplos clientes com múltiplas consultas. Para cada categoria de teste criada, foram realizados testes em máquinas físicas(PM) e em máquinas virtuais(VM), sendo que os tempos obtidos incluem todo o processo de comunicação e processamento. Após os experimentos, os autores identificaram que Cassandra obteve melhor desempenho, fato observado devido a maior capacidade da base Cassandra em lidar com uma quantidade muito grande de dados. A Figura 45 mostra os resultados obtidos nos experimentos com máquinas físicas e virtuais.

Em (ZHANG et al., 2014), os autores realizaram testes experimentais para validar um *framework* por nome de HBaseSpatial, que usa um modelo de armazenamento próprio baseado no HBase, com dados espaciais (incluindo posicionais) nas bases HBaseSpatial, MongoDB e MySQL. Para realizar o experimento, foram utilizados vários *hosts* virtuais através do VirtualBox contendo a base HBase instalada. A configuração dos *hosts* é a seguinte: CPU Duo T7700 com 4GB de memória, Windows 7; máquina virtual com Ubuntu Linux 10. 1, 1GB de memória, Hadoop-0. 20. 203, HBase -0. 90. 3. Ambiente de desenvolvimento Eclipse 3. Os dados de teste foram obtidos da aplicação Jackpine, contendo um total de 1GB de dados espaciais. A Figura 46 mostra os resultados obtidos nos experimentos de inserção com a base HBase.

Para a seção de experimentos de consultas, foram utilizados comparativos de 3 tipos de dados espaciais, *Point data, LineString data e MultiLineString data*, utilizando as bases MongoDB e MySQL, além do HBase. Para cada figura, o eixo horizontal representa a quantidade

Figura 45 – Resultados dos testes de armazenamento do EPCIS

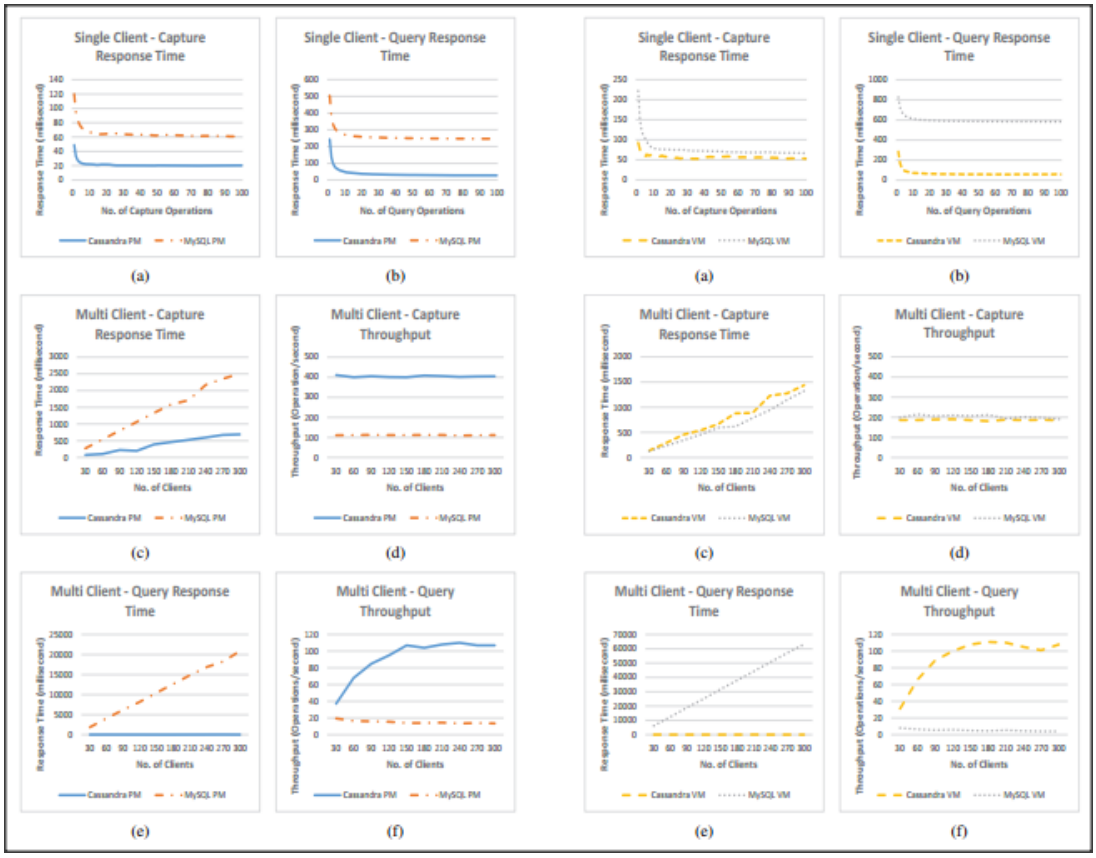


Figura 46 – Resultados dos testes inserção no HBase no HBaseSpatial

Inserting Test(M)	Node 1(s)	Node 5(s)	Node 6(s)	Node 7(s)
6M	3. 112	2. 932	2. 891	2. 883
12M	20. 251	19. 231	18. 911	18. 882
50M	131. 223	109. 231	93. 224	92. 121
100M	214. 324	208. 421	181. 212	180. 993
500M	1231. 231	1107. 122	883. 231	851. 823
1G	2632. 234	2214. 324	1753. 234	1702. 214

de dados consultada e o eixo vertical o tempo em segundos. Como resultado, para a parte de consultas foi constatado que o *framework* proposto pelo autores possui melhor desempenho que as bases MongoDB e MySQL. As Figuras 47, 48, 49 mostra os resultados obtidos nos experimentos de consultas para os dados do tipo *Point, LineString e MultiLineString*, respectivamente.

4.3 Comparativo entre trabalhos da RSL e a dissertação

A Tabela 9 lista todas as características chave que foram utilizadas na revisão sistemática. O item 1 serve para identificar o trabalho relacionado e na última linha o trabalho proposto, a fim de possibilitar uma comparaçao de características entre os trabalhos relacionados e o trabalho proposto. Os demais itens são as características buscadas nos trabalhos relacionados, às quais, juntamente com o item 1, serão colunas da Tabela 10. Os itens 4, 5 e 6 são os mais importantes pois estão diretamente ligados ao que foi proposto no trabalho realizado. Para

Figura 47 – Resultados dos testes de consulta para dados do tipo *Point* no HBaseSpatial

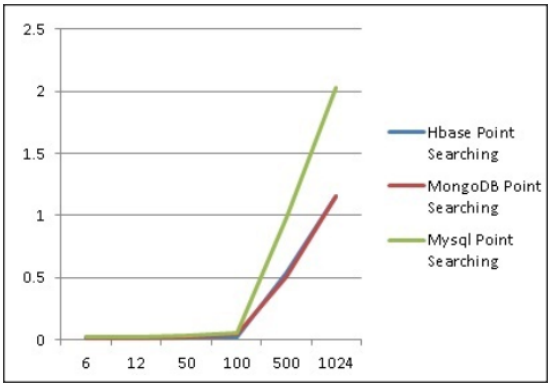


Figura 48 – Resultados dos testes de consulta para dados do tipo *LineString* no HBaseSpatial

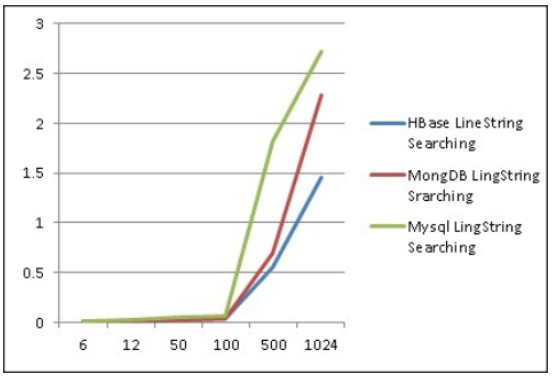
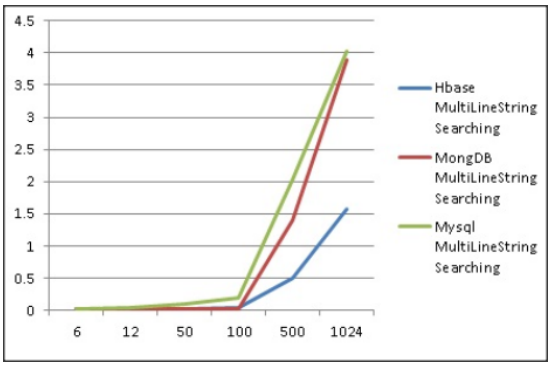


Figura 49 – Resultados dos testes de consulta para dados do tipo *MultiLineString* no HBaseSpatial



identificar arquiteturas já existentes e experimentos de avaliação em NoSQL, tem-se os itens 5 e 6.

A Tabela 10 sumariza o estado da arte e o compara com o trabalho realizado. Nota-se que a maioria dos trabalhos relacionados usam no máximo dois tipos de dados em IoT, no máximo dois tipos de armazenamento NoSQL, além de uns não definirem uma arquitetura ou quando a definem, não realizar experimentos. A última linha da tabela destaca as características do trabalho realizado, de acordo com a Tabela 9. O trabalho realizado, além de definir uma arquitetura, avalia o desempenho de escrita e leitura dos três principais tipos de dados em IoT utilizando os três principais bancos de dados NoSQL. Outro ponto importante fornecido foi uma

Tabela 9 – Características das soluções de armazenamento em IoT existentes

1	Trabalho Relacionado (Trabalho proposto na última linha)
2	Tipo de dado de IoT
2.1	Dados escalares (ex.: temperatura, pressão)
2.2	Dados posicionais (ex.: latitude, longitude)
2.3	Dados multimídia (ex.: videos, fotos)
2.4	Outros dados IoT (ex.: textual)
3	Tipo de armazenamento NoSQL
3.1	Banco de dados de documentos (ex.: MongoDB)
3.2	Banco de dados chave-valor (ex.: Redis)
3.3	Banco de dados de família de colunas (ex.: Cassandra)
4	Dispositivos (ex.: Raspberry Pi, Arduino, sensores)
5	Framework ou arquitetura
6	Experimentos com NoSQL

Tabela 10 – Características dos trabalhos relacionados

1	**2**				**3**			**4**	**5**	**6**
	2.1	**2.2**	**2.3**	**2.4.**	**3.1**	**3.2**	**3.3**			
(LI et al., 2012)	X		X						X	
(SON et al., 2015)	X				X				X	
(FAZIO et al., 2015)			X		X				X	
(HUO et al., 2016)	X					X	X		X	
(LI et al., 2014)							X		X	
(LIN et al., 2014)			X		X				X	
(LIU; WU; LIAN, 2015)			X				X		X	
(LUAN et al., 2015)		X			X				X	
(MA et al., 2012)	X	X			X				X	
(MA; MOTTA; LIU, 2017)	X	X			X				X	
(RONKAINEN; IIVARI, 2015)							X		X	
(YANG; WANG; ZHAO, 2016)	X			X		X			X	
(ZDRAVESKI et al., 2017)					X				X	
(AYDIN; HALLAC; KARAKUS, 2015)		X			X				X	
(FAZIO; PULIAFITO; VILLARI, 2014)					X				X	
(VANELLI et al., 2017)	X						X	X	X	X
(KEBAILI et al., 2016)	X				X			X	X	X
(FRANCESCO et al., 2012)			X		X			X	X	X
(CUNHA; FUKS, 2015)				X	X			X		X
(HASHI et al., 2015)				X	X					X
(HONG et al., 2016)			X		X			X		X
(KANG et al., 2016)				X	X					X
(LU; FANG; LIU, 2016)				X		X			X	X
(MOAWAD et al., 2015)	X		X		X					X
(RADU et al., 2016)				X	X		X			X
(WANG et al., 2016)	X						X	X		X
(ZHAI et al., 2016)	X				X				X	X
(ZHANG et al., 2013)	X					X	X		X	X
(PHAN; NURMINEN; FRANCESCO, 2014)	X		X		X	X				X
(VEEN; WAAIJ; MEIJER, 2012)	X				X		X			X
(LE et al., 2014)	X						X			X
(ZHANG et al., 2014)				X	X		X		X	X
Trabalho Realizado	X	X	X	X	X	X	X	X	X	X

descrição dos diferentes dispositivos responsáveis pela coleta dos dados IoT trabalhados.

5 Proposta de Arquitetura

Como relacionado na seção anterior, várias foram as arquiteturas propostas para armazenamento de dados de IoT e diversos tipos de experimentos foram realizados para avaliar o desempenho entre bases de dados relacionais e não relacionais, em diferentes contextos. Nesta seção, são discorridos a metodologia, a contextualização e o problema que foi tratado nesta dissertação, a definição de uma arquitetura de armazenamento para IoT e o contexto experimental relacionado para validação da arquitetura proposta.

5.1 Metodologia

A seguir, são descritas a caracterização da pesquisa, onde são explicados os tipos de pesquisa que foram realizados e também os procedimentos metodológicos que descrevem os procedimentos utilizados para correta identificação dos trabalhos relacionados ao problema pesquisado bem como o desenvolvimento da pesquisa, definição da arquitetura proposta e sua avaliação.

5.1.1 Caracterização da Pesquisa

A metodologia utilizada neste trabalho foi uma pesquisa qualitativa, quantitativa e experimental que foi organizada em três fases. A primeira fase correspondeu a revisão da literatura, a segunda fase foi o desenvolvimento de uma arquitetura que contextualizou o problema identificado e, por fim, a terceira fase consistiu na realização de experimentos para validação do modelo proposto.

5.1.2 Procedimentos Metodológicos

A revisão da literatura consistiu da realização de uma revisão sistemática contendo conceitos sobre Internet da Coisas, modelos de Internet das Coisas centrados em Nuvem, armazenamento de dados em IoT, técnicas e tipos de armazenamento de dados NoSQL; além da busca por trabalhos correlatos que acrescentem informações a este trabalho, tanto com o fornecimento de características incorporadas, como para futura comparação. O foco da revisão sistemática foi em identificar trabalhos que contextualizassem o problema de indefinição sobre o tipo mais adequado de armazenamento de dados de sensores de IoT, além da busca por trabalhos que realizaram experimentos utilizando armazenamento NoSQL em IoT com relação ao tempo de inserção e recuperação dos dados.

Na fase de desenvolvimento foi definida uma arquitetura híbrida de alto nível que contextualiza o problema de indefinição de armazenamento escalável de dados de sensores de

IoT. A arquitetura contempla camadas e componentes que foram, em parte, operacionalizados na fase de realização de experimentos.

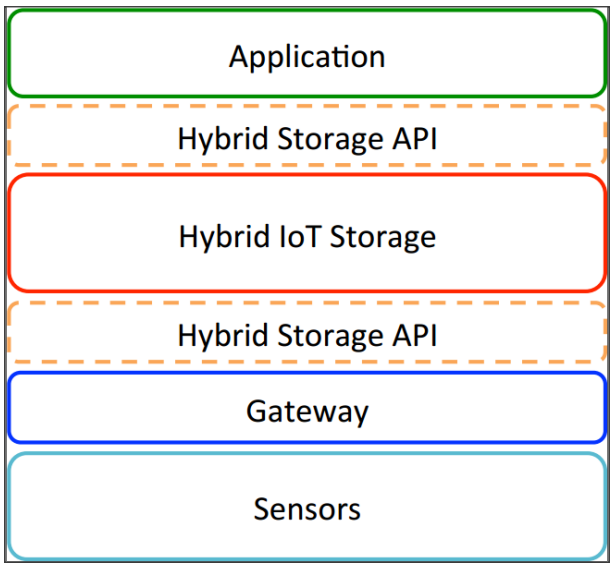
Em seguida, foi realizada a implementação da fase de Execução e experimentação do trabalho com a configuração do ambiente necessário para o experimento, onde cada base de dados foi configurada através de APIs desenvolvidas. Estas APis também foram responsáveis pela geração da massa de dados necessária para o experimento. Todo o ambiente foi configurado e os experimentos realizados utilizando o mesmo hardware, de forma a ter um um experimento com condições uniformes entre os modelos testados.

5.2 Contextualização e Definição do Problema

Considerando que dados escalares, multimídia e posicionais estão entre os principais tipos de dados em IoT (PHAN; NURMINEN; FRANCESCO, 2014), temos como problema determinar qual o armazenamento adequado para cada um desses tipos. Apesar de ciente da existência de abordagens SQL (STONEBRAKER, 2010), NewSQL (ASLETT, 2011; CETINTEMEL et al., 2014) e sistemas de arquivos em larga escala baseados em nuvem (ZHAO et al., 2014; BESSANI et al., 2014), a abordagem NoSQL se mostra mais adequada para essa tarefa (FOWLER, 2015).

Para direcionar a solução do problema levantado, foi definida uma arquitetura híbrida em camadas para representar o modelo de armazenamento não relacional, referenciado pela Figura 50. Um modelo em camadas foi escolhido por representar melhor o fluxo de dados dos sensores em meio à arquitetura, além de facilitar a divisão das responsabilidades existentes em cada componente da arquitetura, principalmente em relação às camada relativas ao armazenamento de dados, pois representam o foco do trabalho.

Figura 50 – Arquitetura Híbrida em Camadas



No modelo proposto, existem 4 camadas principais e 2 camadas intermediárias, chamadas

de *Hybrid Storage APIs*. A ideia base é que cada camada possua sua função bem definida na arquitetura, onde o dado seguirá o fluxo descrito.

Nela, a camada de sensores tem por função conter todos os sensores que serão responsáveis por captar diversos tipos de dados, como temperatura, pressão, umidade, fotos, vídeos, etc.; ou seja, conterá todos os dispositivos necessários para captação de dados brutos. Nessa camada, é possível haver a presença de diversas forma de sensoriamento, com vários tipos distintos de sensores, como por exemplo, o sensor de variação de temperatura ADT7320, o sensor de variação de pressão LPS25H, o sensor de variação de umidade HTU21D. Além destes sensores, há a possibilidade que um ou mais desses sensores estejam alocados em uma *protoboard* multifuncional como *Arduino*, *Raspberry Pi* ou *BeagleBone*, por exemplo.

A camada *Gateway* possui os *gateways* para obter os dados dos dispositivos. Um gateway é um elemento comum dentro de um domínio IoT, e assumimos que ele mantém suas funções tradicionais, como agregação, filtragem e segurança. Uma infraestrutura de IoT possui um fluxo de dados bem definido. Normalmente começa com sensores enviando dados para uma estação base através de *gateways*. Esses *gateways* possuem diferentes funções, como a filtragem de dados de modo seguro, fornecendo confidencialidade e integridade, por exemplo. Em seguida, esses *gateways* enviam os dados para uma base de dados centralizada onde as aplicações podem obter e processar esses dados. Em nosso fluxo de arquitetura, um *gateway* recebe os dados dos sensores e encaminha para a camada de armazenamento. Com isso, a camada de gateway fornece a interoperabilidade entre sensores e o armazenamento híbrido. Em nossa arquitetura, o gateway precisa usar a API de armazenamento para enviar os dados coletados para o sistema de armazenamento híbrido.

A camada de API híbrida é onde *middlewares* estarão presentes para coletar os dados tratados pelos *gateways* e possibilitar seu correto envio para a camada de armazenamento. Partindo-se da premissa que cada tipo de dado possua um modelo de armazenamento mais adequado, essa camada será responsável por direcionar cada tipo de dado para seu modelo de armazenamento adequado. Dado o direcionamento do dado e de em qual modelo de armazenamento o mesmo deve ser persistido, a camada armazenamento contém os tipos de bancos de dados NoSQL para armazenar os dados recebidos pela API híbrida. A API fornece a cola entre gateways IoT e o sistema de armazenamento híbrido. Diversas tecnologias podem ser utilizadas para permitir a criação dessa API para coletar, tratar e direcionar os dados obtidos, como por exemplo, as linguagens: Python, Java, Javascript, C# (plataforma .NET); além do uso de padrões como *JSON*. O sistema de armazenamento híbrido é composto de diferentes motores NoSQL e tipos de armazenamento de dados (por exemplo, Redis e valor-chave).

No entanto, vale a pena enfatizar que sensores, gateways e aplicativos externos não estão cientes dessa diversidade de mecanismos de armazenamento. Eles meramente usam a API híbrida. Na prática, diferentes tipos de dados de IoT serão armazenados de forma automática e transparente em diferentes bases de dados NoSQL. Por exemplo, valores de umidade dos

sensores DHT11 serão armazenadas no banco de dados Redis. Por outro lado, objetos multimídia serão armazenados em Cassandra.Resumidamente, o papel principal da API híbrida é identificar os tipos de dados enviados pelos gateways e armazená-los no mais adequado banco de dados. Além disso, outra função da API é fornecer mecanismos de armazenamento com foco em *Big Data* para aplicativos externos.

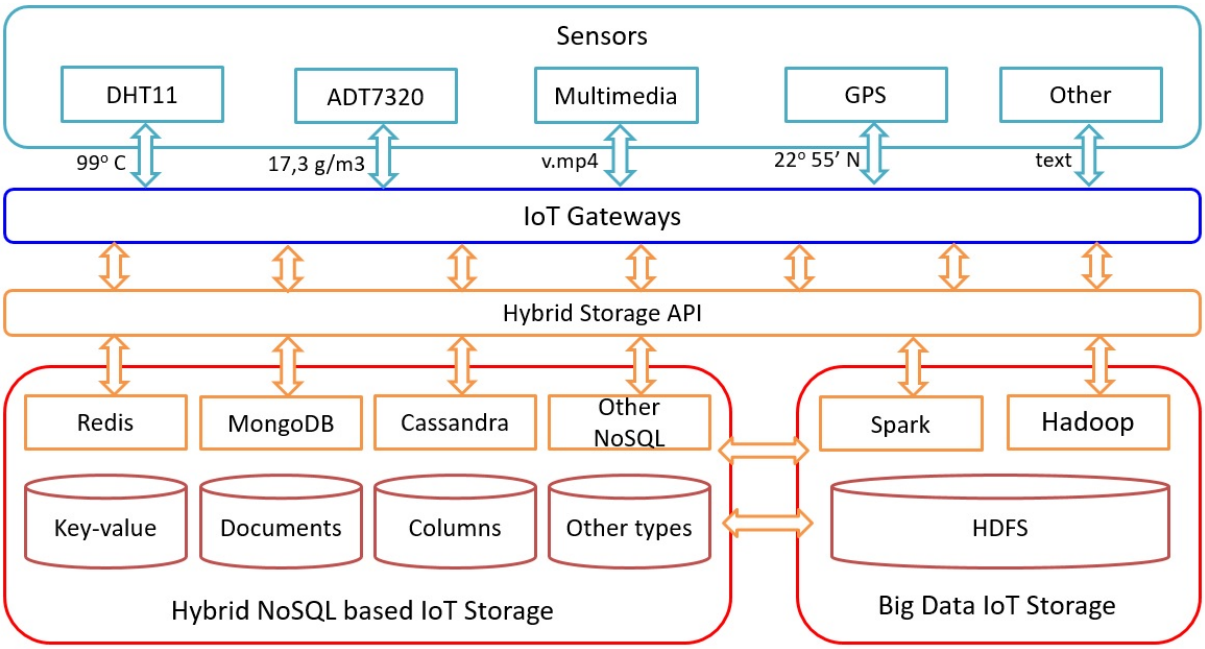
Devido à necessidade prover o mínimo de perda de dados e de prover retorno eficiente em aplicações críticas de sensores, o principal critério arquitetural para escolha do local de armazenamento é o desempenho do tipo de dado em cada banco de dados. Se um banco de dados NoSQL de valor-chave, como o Redis, for identificado como o mecanismo de melhor desempenho para armazenar dados escalares, ele o fará. No entanto, é importante frisar que as aplicações podem ter necessidades específicas, como usar solicitações *JSON*, por exemplo. Com isso, uma aplicação pode demandar um modelo de armazenamento que lide com esse tipo de solicitação. Como exemplo, o banco de dados MongoDB lida nativamente com esse tipo de requisição. A arquitetura proposta possibilita essa definição por parte das aplicações. Em relação ao critério primário de desempenho na leitura e escrita dos dados, para habilitar a API de armazenamento a tomar uma decisão na escolha de qual banco de dados NoSQL deve ser escolhido, experimentos foram realizados para identificar qual é o mecanismo NoSQL de melhor desempenho para cada tipo de dado trabalhado. A seção experimental deste trabalho cuida desta parte.

Além de definir qual é o mecanismo adequado para armazenar os tipos de dados e, considerando que milhões de dados não relacionados serão coletados pelos dispositivos, é importante fornecer funções de análise de dados, como a mineração de dados e análises *Big Data*. *Frameworks* como Hadoop (TUTORIALSPOINT, 2018a) e Spark (APACHE, 2018d) podem ser utilizados. Também é importante fornecer um canal de comunicação entre a API de armazenamento e as aplicações, a fim de permitir que os aplicativos possam definir critérios de armazenamento para a arquitetura de forma parametrizada. A camada de armazenamento híbrido é a camada mais crucial do arquitetura. Em suma, a camada de armazenamento tem duas funções. Primeiro, uma API de armazenamento que pode ser usada por gateways e aplicativos para armazenar e recuperar dados. Esta API esconde técnicas e específicas aspectos de qualquer sistema de banco de dados NoSQL. Em segundo lugar, a camada é composta por diferentes sistemas de banco de dados NoSQL que fornecem armazenamento escalável e eficiente para diferentes tipos de dados não estruturados, como documentos (por exemplo, MongoDB), colunas (por exemplo, Cassandra), valor-chave (por exemplo, Redis) e assim por diante. Com isso, todos os gateways e aplicativos IoT dependem dessa camada.

Por fim, a camada de aplicações é a responsável por obter os dados da camada de armazenamento e fornecer visualização e uso desses dados por aplicações externas, possibilitando o uso dos dados de forma aplicada por diversas áreas: Engenharia, Medicina, Segurança, etc. Essa camada consome os dados produzidos pelos dispositivos (por exemplo, sensores de temperatura

ou umidade). Em relação ao uso pelas aplicações, um sistema de controle de nivelamento de óleo em uma fábrica pode precisar receber os dados de todos os sensores espalhados em caldeiras para identificar automaticamente a existência de vazamentos a fim de manter a operação da fábrica e os operadores sempre em funcionamento. Neste caso, o sistema monitora os sensores da caldeira em tempo real. Além disso, uma aplicação pode consumir dados de muitos dispositivos de um ou mais domínios IoT. Assim, escalonamento e desempenho são requisitos críticos para suporte a diferentes tipos de aplicações em IoT. Também é importante mencionar que as aplicações podem gravar dados para o armazenamento que podem ser usados por razões históricas (por exemplo, mineração de dados e análises de *Big Data*) ou para compartilhar dados processados com outras aplicações. A Figura 51 mostra detalhes de cada camada da arquitetura.

Figura 51 – Arquitetura Híbrida em Camadas detalhada



5.2.1 Operações da Arquitetura Híbrida

A API de armazenamento envia e responde solicitações no formato *JSON* por meio de um conjunto de operações para permitir a comunicação entre a camada de armazenamento e os aplicativos externos. As operações e suas definições são:

- setData: armazena dados.
- getData: obtem dados.
- defineStorageOption: Define o critério de armazenamento.

Além de definir as operações, é necessário especificar os atributos e seus valores em uma solicitação. Os possíveis atributos e suas definições são:

- operation: Indica a operação.
- db: Especifica o nome do banco de dados para armazenar dados. Cria se não existe.
- storageOption: Especifica o critério de armazenamento. As opções de valor são "performance", que é o valor padrão, e "document", se é necessário um banco de dados de documentos que trate de solicitações nativamente *JSON*.
- dataAnalysis: Especifica se usará bancos de dados para análise de big data. Se não especificado, não será usado. "spark"e "hadoop"são opções válidas. Se não especificado, não será usado.

O exemplo a seguir é uma solicitação simples para armazenar dados por meio da operação "setData".

```
{
        "operation" : "setData",
        "db" : "temperature_db",
        "data" : "56",
        "storageOption:" : "performance",
        "dataAnalysis" : "spark"
}
```

Este capítulo realizou a descrição da arquitetura definida, suas características, as operações possíveis, além das funcionalidades já desenvolvidas. É importante ressaltar que uma das principais funções desta arquitetura é ocultar detalhes técnicos do armazenamento dos dados para as aplicações consumidoras, ou seja, prover um armazenamento transparente, independente do tipo do dado captado. A arquitetura criada é expansível e pode lidar com o uso de outros tipos de dados de IoT não trabalhados nesta dissertação, novas operações, além de prover armazenamento para outros bancos de dados NoSQL, características que serão pesquisadas em trabalhos futuros e são melhor detalhados no último capítulo desta dissertação.

6 Avaliação e Resultados Experimentais

Para a correta validação da arquitetura híbrida proposta, foram realizados experimentos com foco na avaliação do tempo de leitura e escrita de dados escalares, posicionais e multimídia, tipos principais de dados de sensores em IoT. Para armazenar esses dados, foram utilizados os bancos de dados NoSQL Redis (chave-valor), MongoDB (banco de dados de documentos), Cassandra (família de colunas) e Neo4j (grafos). Vale destacar que, entre os experimentos já realizados em trabalhos anteriores, não se identificou que algum desses projetos tivesse como foco experimentos de avaliação de desempenho entre os principais bancos de dados NoSQL existentes, ou, quando realizados, foi de forma parcial, contemplando apenas alguns tipos dessas bases. Essas bases de dados estão listadas num *ranking* de popularidade de bancos de dados (IT, 2018). Vale ressaltar que os experimentos foram realizados com tanto com armazenamento em disco como em memoria para todos os bancos de dados, objetivando verificar se o modelo em memória supera o modelo em disco, proporcionando uma validação mais completa em relação à análise de desempenho.

Dentre os bancos de dados NoSQL testados, é possível adiantar que entre os bancos Redis, MongoDB e Cassandra, o Cassandra teve o pior desempenho nas leituras e escritas. Seguindo, ao realizar testes de escrita de dados escalares com o Neo4j, verificou que este teve em média 6x pior desempenho que o banco de dados Cassandra. Devido a esse baixo desempenho, e também ao objetivo básico dos experimentos ser buscar qual o tipo armazenamento possui melhor desempenho em cada tipo de dado, decidiu-se por abandonar o uso do Neoj nos demais experimentos.

6.1 Configuração do Experimento

A seguir são informados os itens necessários para a execução do experimento, como a configuração do ambiente necessário, a geração da massa dos dados utilizada nos experimentos e a organização dos casos de testes simulando cada cenário.

6.1.1 Ambiente e Massa de Dados

Para a realização dos testes foi configurado um ambiente contendo servidores locais para cada banco de dados NoSQL. A máquina usada possui um Sistema Operacional Lubuntu versão 16.04 de 64 bits, contendo um processador Intel Core i7-4510 com uma CPU de 2,60 GHz, 8 GB de memória e 100Gb de armazenamento.

A avaliação foi realizada usando conjuntos de dados de 1MB, 2MB, 4MB, 8MB, 16MB, 32MB e 64MB, para cada um dos tipos de dados. Dados escalares (por exemplo, temperatura

40°C), dados posicionais (por exemplo, latitude 26°60'N) e dados multimídia (por exemplo, um pedaço de um filme com 64mb de tamanho), foram gerados aleatoriamente através de módulos de uma API Python desenvolvida. Os módulos da API em Python também foram usados para executar a avaliação, ou seja, inserir e consultar dados do armazenamento híbrido. Todo o código fonte do experimento está disponível em <https://github.com/brauliolivio/SensorNoSQLStorage>. Vale a pena enfatizar que conjuntos de dados similares foram usados em (PHAN; NURMINEN; FRANCESCO, 2014).

6.1.2 Casos dos Testes

Os testes foram realizados com 7 conjuntos de dados, 3 tipos diferentes de dados, 2 modalidades de armazenamento (memória e disco) e 2 operações (escrita e leitura de dados). Um exemplo prático de caso de teste seria: Avaliar a inserção de 64Mb de dados posicionais com o banco de dados Cassandra em memória. Considerando as combinações realizadas para todos os casos possíveis, têm-se um total de 84 casos de testes no experimento.

A fim de prover maior validade ao experimento, cada caso de teste foi repetido um total de dez vezes. Com isso, foram realizadas ao todo 840 execuções no experimento. Vale salientar também que os valores contidos nos gráficos dos resultados representam a média das dez execuções por caso de teste. Ainda, é importante destacar que execuções maiores que os dez casos do experimento apresentaram poucas variações em relação aos resultados encontrados. Com isso, é possível concluir que o aumento no número de execuções não influenciou no resultado final do experimento para mais de dez execuções.

6.2 Análise estatística dos dados

Nesta seção estão descritos os indicadores estatísticos utilizados para analisar os tempos retornados por cada banco de dados nas operações em cada conjunto de dados.

6.2.1 Hipóteses

Considerando a análise estatística sobre os dados obtidos, é possível elencar duas hipóteses: uma hipótese principal, que indica que à medida que o tamanho do conjunto de dados aumenta, os tempos de cada banco de dados tornam-se mais dispersos em relação à media de tempo do grupo (H0). Uma hipótese alternativa, que indica que não há correlação entre o aumento no tamanho do conjunto de dados e o nível de dispersão dos dados em cada grupo (Ha).

6.2.2 Indicadores estatísticos

Foram utilizados como indicadores o desvio padrão (s) e o coeficiente de variação percentual (c.v.%) entre os tempos retornados pelos bancos de dados nas operações. Os dois

indicadores são utilizados para medir o nível de dispersão dos dados em torno da média ($\bar{\mathbf{x}}$). Com isso, quanto maior o desvio padrão, maior a dispersão dos dados retornados em relação à media do grupo. Da mesma forma, Quanto menor o valor do coeficiente de variação percentual, mais os dados estarão concentrados em torno da média (ISOTALO, 2001).

6.2.3 Objetivo na análise

Como objetivo principal, buscou-se avaliar a variação do desvio padrão entre os conjuntos de dados em cada grupo (1MB a 64MB) para confirmar ou rejeitar a hipótese H0. Também buscou-se avaliar a diferença existente entre o coeficiente de variação percentual calculado em cada grupo, igualmente para confirmar ou rejeitar a hipótese H0.

6.3 Resultados

A seguir serão mostrados os gráficos com os resultados do testes de inserção e leitura de dados escalares, posicionais e multimídia em memória e em disco para cada conjunto de dados. Verificou-se que a hipótese H0 não foi confirmada, já que a dispersão dos dados nem sempre aumentava quando o tamanho do conjunto de dados aumentava. Em alguns casos, conjuntos menores obtiveram maior dispersão que conjuntos maiores. Além dos gráficos, foram descritos os percentuais de desempenho em cada comparação entre os bancos de dados. Para cada caso de teste, o valor percentual foi obtido através da diferença de tempo entre dois bancos de dados diferentes. Como exemplo, supondo que o tempo de inserção para 1MB de dados escalares seja 5s para Redis em memória e 34s para o MongoDB em memória, a variação entre esses dois valores pode ser calculada pela equação $((T2/T1)-1)*100$. Como resultado, o Redis em memória superou o MongoDB em memória para o conjunto de dados de 1MB. Neste cenário de exemplo, após repetir este cálculo para cada um dos conjuntos de dados(1Mb a 64Mb), e calcular a média aritmética simples entre eles, obtém-se um ganho médio do Redis em relação ao MongoDB em memória de 578% para dados escalares.

6.3.1 Inserção de dados

Nesta seção, serão mostrados os resultados dos testes das inserções de dados escalares, posicionais e multimídia nos bancos Redis, Cassandra e MongoDB, tanto para os bancos configurados em memória como para os configurados com armazenamento em disco.

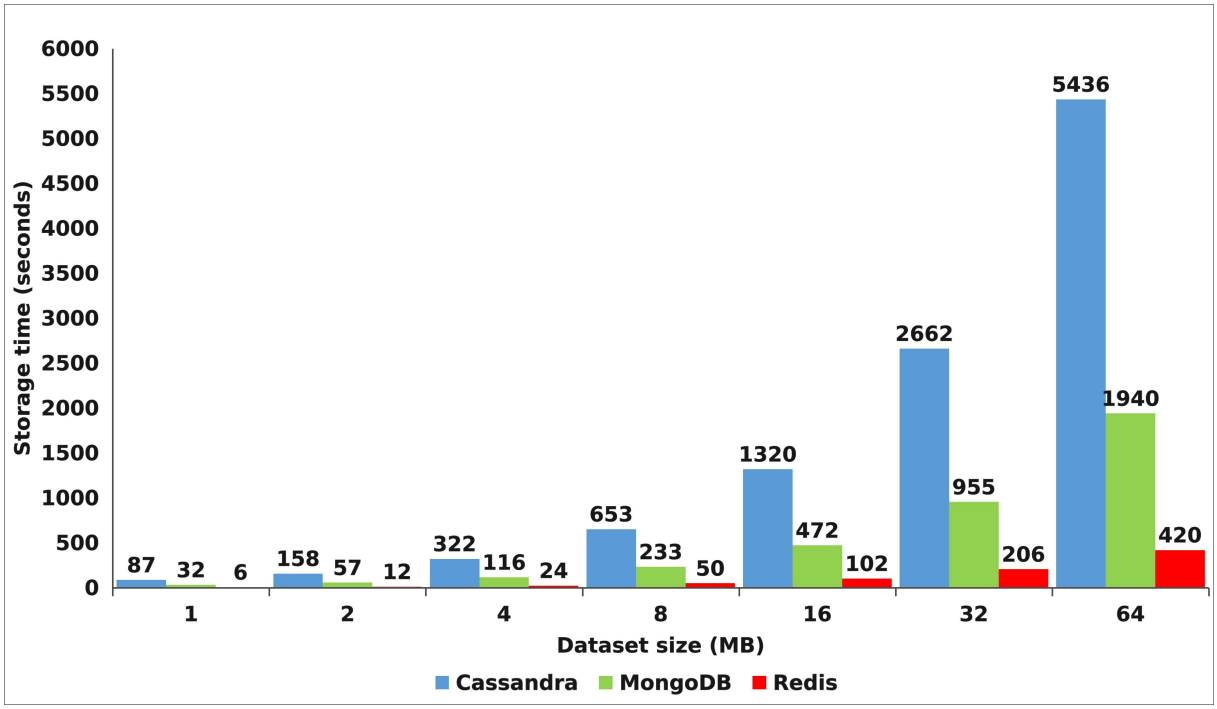
Devido a um dado escalar e um dado posicional possuirem em média 1 byte, os conjuntos de dados escalares e posicionais foram gerados pela API em python de forma aleatória até atingir o tamanho desejado para os casos de testes. Exemplo: Para inserção com 64Mb de dados escalares, a API em python gera valores aleatórios de temperaturas (tipo escalares) de 0 a 100 até o conjunto formar 64Mb. Com o conjunto possuindo o tamanho desejado, bastou executar a inserção de cada caso de teste.

Para definir os conjuntos de dados multimídia, a estratégia foi diferente em relação aos dados escalares e posicionais. Como é comum encontrar arquivos multimídia maiores que tamanho máximo testado, que foi de 64Mb, a estratégia foi dividir arquivos de vídeo nos conjuntos de dados necessário. Com isso, um arquivo de vídeo maior de 64Mb foi dividido em pedações menores contendo: 1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb e 64Mb. Além disso, cada arquivo de vídeo foi divido em pedaços iguais de 1024 bytes, onde cada pedaço foi individualmente inserido até atingir o total referente ao tamanho do conjunto. Essa divisão foi necessária para validar testes de consulta mais próximos da realidade, onde um dado deve ser encontrado num conjunto maior.

6.3.1.1 Inserção de dados escalares em memória

Os primeiros testes realizados foram de inserção de dados escalares em memória, ou seja, cada banco de dados testado foi configurado para gravar apenas em memória. O Gráfico 52 exibe os resultados encontrados.

Figura 52 – Desempenho de inserção de dados escalares em memória



Após as 10 execuções de cada conjunto de dados com cada banco de dados, é possível notar que o Redis obteve menor tempo de inserção que o MongoDB e o Cassandra em todos os conjuntos. O Redis obteve em média 314% de desempenho superior ao banco de dados MongoDB e em média 1171% de desempenho superior ao banco de dados Cassandra. A Tabela 11 indica os resultados estatísticos encontrados. É possível notar que, devido aos valores de coeficiente de variação aproximados em todos os grupos, pode-se dizer que todos os grupos

possuem dispersão aproximada. Contudo, apesar da leve variação, os grupos de 1Mb e 4Mb foram os que apresentaram maior dispersão dos dados em relação à média.

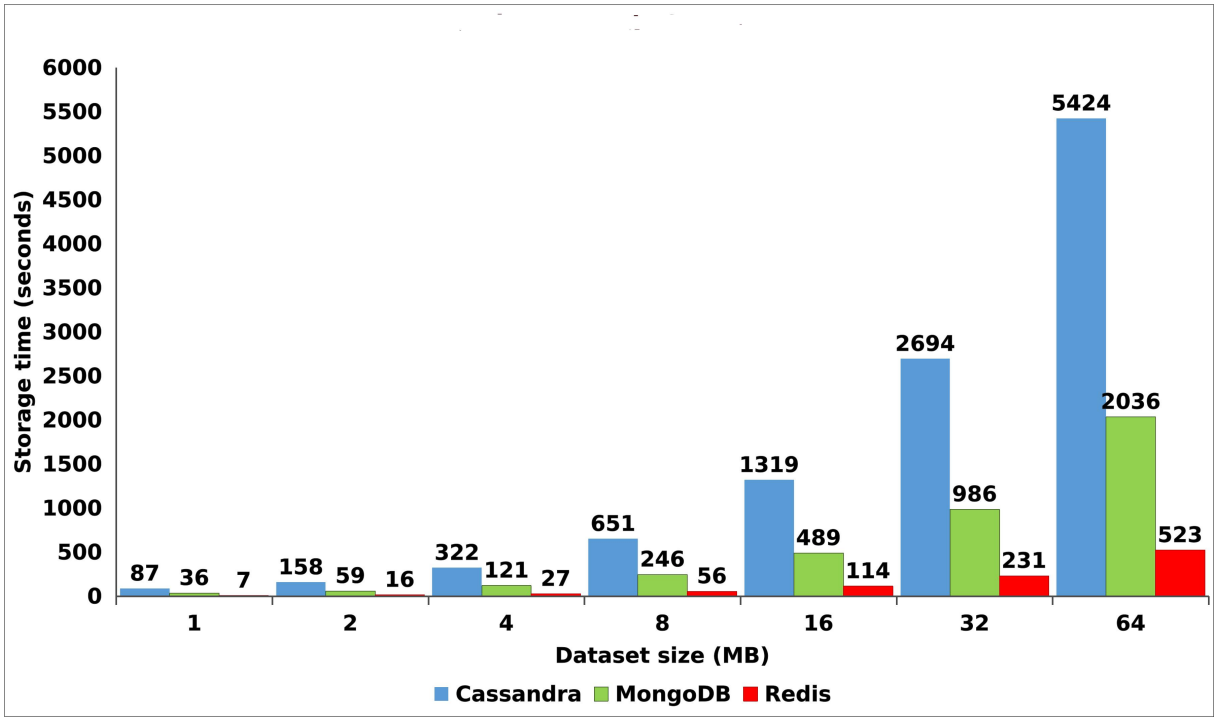
Tabela 11 – Resultado estatístico da inserção escalar em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	6, 32, 87	41.67	41.35	0.81
2Mb	12, 57, 158	75.66	74.76	0.80
4Mb	24, 116, 322	155.33	154.40	0.81
8Mb	50, 233, 653	312	309.16	0.80
16Mb	102, 472, 1320	631.33	624.43	0.80
32Mb	206, 955, 2662	1274.33	1258.75	0.80
64Mb	420, 1940, 5436	2598.67	2572.05	0.80

6.3.1.2 Inserção de dados escalares em disco

A segunda rodada objetivou testar a inserção de dados escalares em disco, ou seja, cada banco de dados testado foi configurado para gravar diretamente no disco. O Gráfico 53 exibe os resultados encontrados.

Figura 53 – Desempenho de inserção de dados escalares em disco



Após as 10 execuções de cada conjunto de dados com cada banco de dados, é possível notar que o Redis obteve menor tempo de inserção que o MongoDB e Cassandra em todos os conjuntos de dados. O Redis obteve em média 285% de desempenho superior ao banco de dados MongoDB e em média 971% de desempenho superior ao banco de dados Cassandra. A Tabela 12 indica os resultados estatísticos encontrados. O grupo de 32Mb foi o que apresentou maior dispersão dos dados em relação à média.

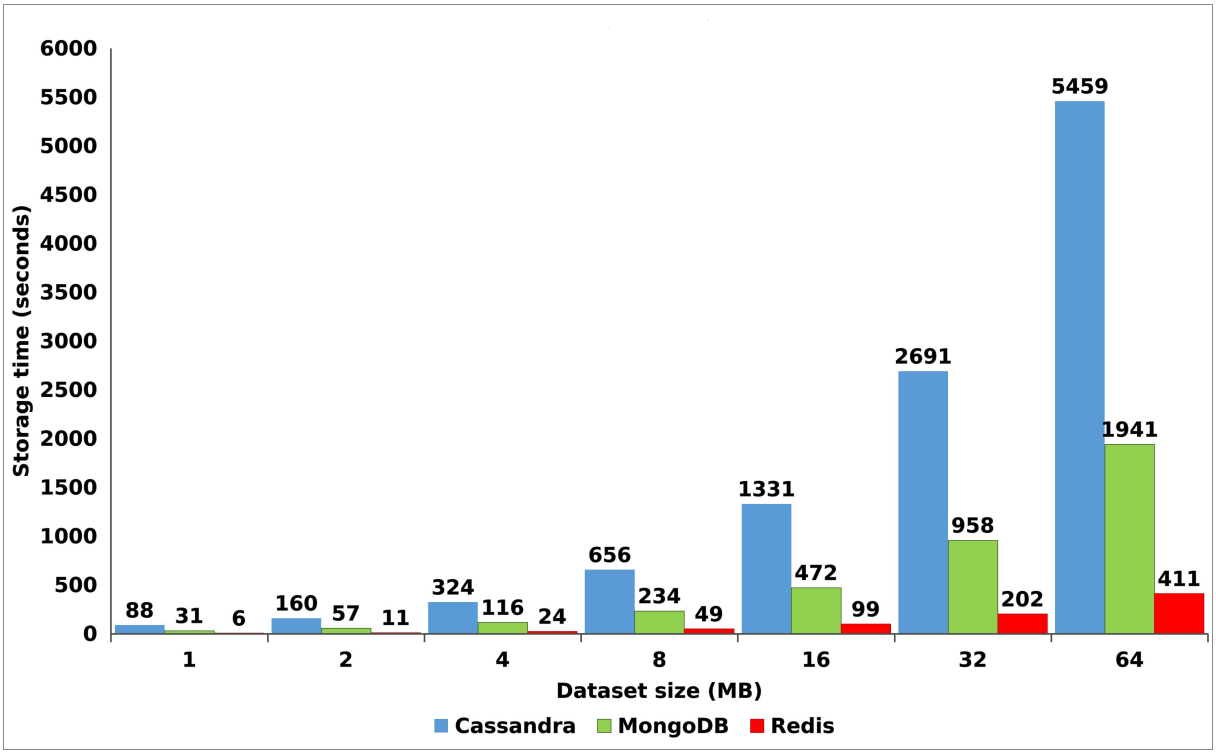
Tabela 12 – Resultado estatístico da inserção escalar em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	7, 36, 87	43.33	40.50	0.76
2Mb	16, 59, 158	77.66	72.81	0.76
4Mb	27, 121, 322	156.66	150.69	0.78
8Mb	56, 246, 651	317.66	303.90	0.78
16Mb	114, 489, 1319	640.66	616.65	0.78
32Mb	231, 986, 2694	1303.66	1261.85	0.79
64Mb	523, 2036, 5424	2661	2509.56	0.77

6.3.1.3 Inserção de dados posicionais em memória

O Gráfico 54 exibe os resultados encontrados na inserção de dados posicionais em memória.

Figura 54 – Desempenho de inserção de dados posicionais em memória



Após as 10 execuções de cada conjunto de dados com cada banco de dados, novamente, o Redis superou os demais bancos de dados. O Redis obteve em média 328% de desempenho superior ao banco de dados MongoDB e em média 1228% de desempenho superior ao banco de dados Cassandra. A Tabela 13 indica os resultados estatísticos encontrados. O grupo de 1Mb foi o que apresentou maior dispersão dos dados em relação à média.

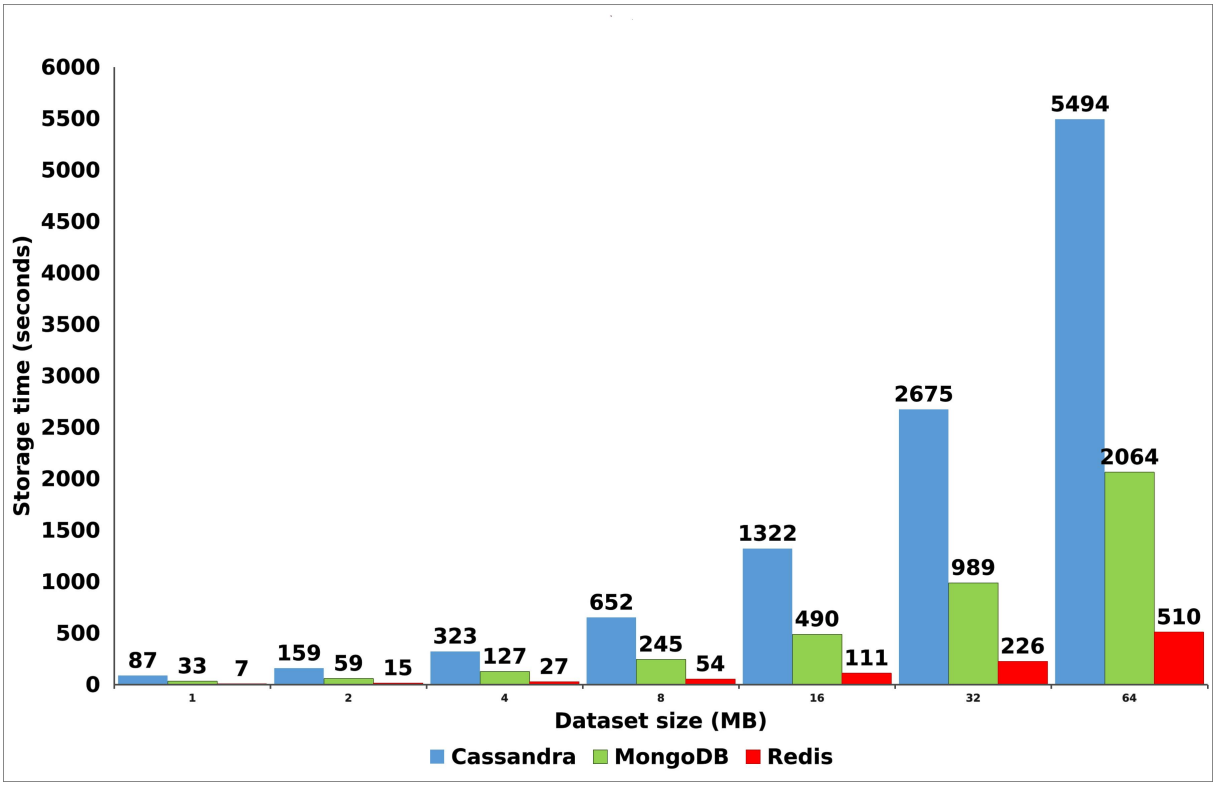
6.3.1.4 Inserção de dados posicionais em disco

O Gráfico 55 exibe os resultados encontrados na inserção de dados posicionais em disco.

Tabela 13 – Resultado estatístico da inserção posicional em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	6, 31, 88	41.67	42.02	0.82
2Mb	11, 57, 160	76	76.29	0.81
4Mb	24, 116, 324	154.67	153.69	0.81
8Mb	49, 234, 656	313	311.11	0.81
16Mb	99, 472, 1331	634	631.77	0.81
32Mb	202, 958, 2691	1283.67	1276.05	0.81
64Mb	411, 1941, 5459	2603.67	2588.42	0.81

Figura 55 – Desempenho de inserção de dados posicionais em disco



Após as 10 execuções de cada conjunto de dados com cada banco de dados, o Redis superou os demais bancos de dados na inserção posicional em disco. O Redis obteve em média 285% de desempenho superior ao banco de dados MongoDB e em média 1000% de desempenho superior ao banco de dados Cassandra. A Tabela 14 indica os resultados estatísticos encontrados. O grupo de 64Mb foi o que apresentou maior dispersão dos dados em relação à média.

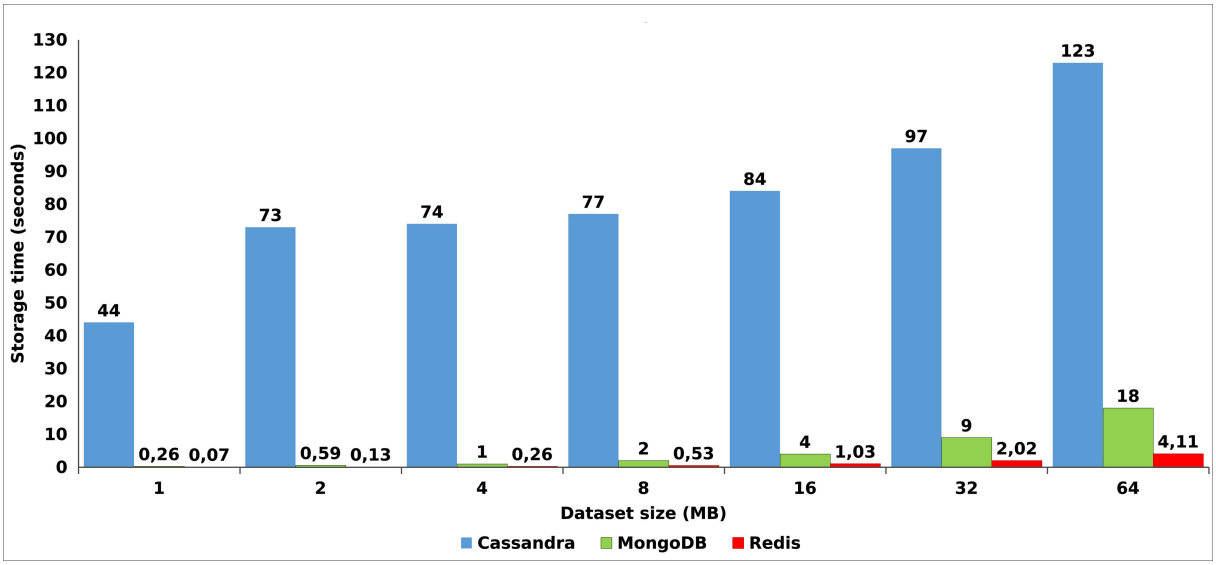
Tabela 14 – Resultado estatístico da inserção posicional em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	7, 33, 87	42.33	40.80	0.78
2Mb	15, 59, 159	77.66	73.79	0.77
4Mb	27, 127, 323	159	150.57	0.77
8Mb	54, 245, 652	317	305.43	0.78
16Mb	111, 490, 1322	641	619.46	0.78
32Mb	226, 989, 2675	1296.66	1253.15	0.78
64Mb	510, 2064, 5494	2689.33	2550.16	0.77

6.3.1.5 Inserção de dados multimídia em memória

O Gráfico 56 exibe os resultados encontrados na inserção de dados multimídia em memória.

Figura 56 – Desempenho de inserção de dados multimídia em memória



Ao fim das 10 execuções de cada conjunto de dados, o Redis obteve, mais uma vez, superioridade em relação aos demais bancos de dados. O Redis obteve em média 308% de desempenho superior ao banco de dados MongoDB e em média 25321% de desempenho superior ao banco de dados Cassandra. É importante notar que, diferente das inserções escalares e posicionais, o Cassandra obteve um desempenho muito inferior aos demais bancos na inserção de dados multimídia, considerando que todos executaram o experimento em igualdade de condições, ou seja, utilizando os mesmos critérios e rodando no mesmo ambiente de execução e de forma isolada para cada cenário. Ao se usar como exemplo o maior conjunto, de 64Mb, e, considerando que cada pedaço de vídeo possuiu 1024 bytes, cada banco realizou um total de 65536 inserções até completar o tamanho necessário do arquivo nesse caso de teste. Dado o aumento exponencial no número de inserções, é provável que o Cassandra tenha sofrido o maior impacto negativo com esse aumento, o que causou maior latência até o fim do experimento. A Tabela 15 indica os resultados estatísticos encontrados. Como o grupo de 64Mb apresentou o maior desvio padrão em relação à média, este é o que possui maior dispersão dos dados.

6.3.1.6 Inserção de dados multimídia em disco

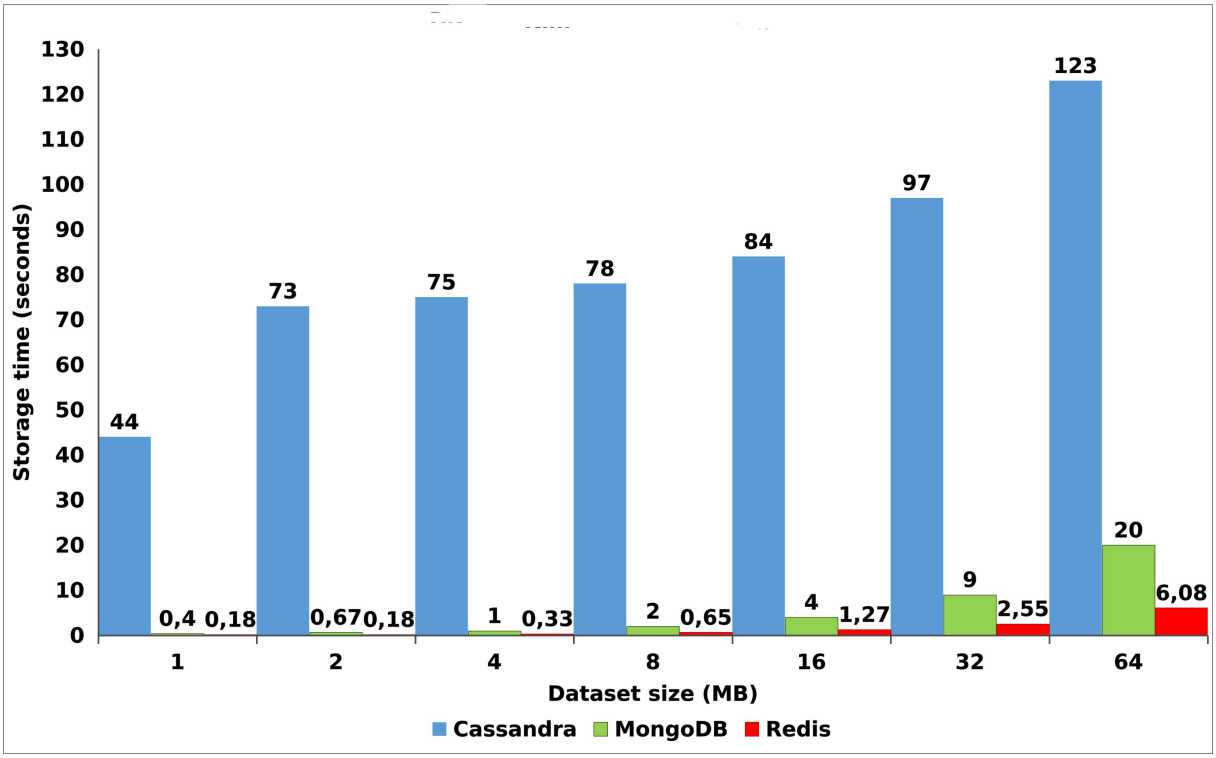
O Gráfico 57 exibe os resultados encontrados na inserção de dados multimídia em disco.

Após as 10 execuções, igualmente aos resultados da inserção multimídia em memória, o Redis também superou os demais bancos de dados na inserção de dados multimídia em disco. O Redis obteve em média 214% de desempenho superior ao banco de dados MongoDB e em média

Tabela 15 – Resultado estatístico da inserção multimídia em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.07, 0.26, 44	14.77	25.30	1.39
2Mb	0.13, 0.59, 73	24.57	41.93	1.39
4Mb	0.26, 1, 74	25.08	42.36	1.37
8Mb	0.53, 2, 77	26.51	43.73	1.34
16Mb	1.03, 4, 84	29.67	47.06	1.29
32Mb	2.02, 9, 97	36	52.93	1.20
64Mb	4.11, 18, 123	48.37	65	1.09

Figura 57 – Desempenho de inserção de dados multimídia em disco



15924% de desempenho superior ao banco de dados Cassandra. A Tabela 16 indica os resultados estatísticos encontrados. Os grupos de 1Mb e 2Mb apresentaram maior dispersão dos dados.

Tabela 16 – Resultado estatístico da inserção multimídia em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.18, 0.40, 44	14.86	25.23	1.38
2Mb	0.18, 0.67, 73	24.61	41.90	1.38
4Mb	0.33, 1, 75	25.44	42.91	1.37
8Mb	0.65, 2, 78	26.88	44.27	1.34
16Mb	1.27, 4, 84	29.75	46.99	1.28
32Mb	2.55, 9, 97	36.18	52.76	1.19
64Mb	6.08, 20, 123	49.69	63.86	1.04

6.3.2 Leitura de dados

Nesta seção, serão mostrados os resultados dos testes de leitura de dados escalares, posicionais e multimídia nos bancos Redis, Cassandra e MongoDB, tanto em memória como em disco.

A estratégia utilizada para leitura de dados escalares e posicionais foi a de buscar um dado específico em cada conjunto de dados. Exemplo: Para leitura de um dado escalar em um conjunto de 64Mb, bastou definir uma temperatura base a ser buscada em meio a um conjunto de temperaturas com tamanho de 64Mb. O único cuidado necessário em cada caso foi garantir que o dado procurado tenha apenas uma única ocorrência no grupo.

Diferente dos testes de inserção, para os testes de leituras, a estratégia para dados multimídia foi a mesma dos dados escalares e posicionais, ou seja, buscar um pedaço pré-definido de um arquivo de vídeo em meio a conjunto de pequenos arquivos de vídeo contendo o tamanho necessário para o teste. Exemplo: Para consultar um arquivo de vídeo em meio a 64Mb de arquivos de vídeo de tamanho igual ao buscado, foi necessário antes particionar um arquivo maior em pedações iguais e menores, além de garantir que o arquivo base que foi buscado tivesse apenas uma única ocorrência no conjunto.

6.3.2.1 Leitura de dados escalares em memória

Os primeiros testes de leitura realizados foram para escalares em memória. O Gráfico 58 exibe os resultados encontrados.

Após as 10 leituras de cada conjunto de dados com cada banco de dados, é possível notar que o Redis obteve menor tempo de busca que o MongoDB e Cassandra em todos os conjuntos de dados. O Redis obteve em média 90354% de desempenho superior ao banco de dados MongoDB e em média 558762% de desempenho superior ao banco de dados Cassandra. A Tabela 17 indica os resultados estatísticos encontrados. O grupo de 64Mb apresentou maior dispersão dos dados por possuir maior desvio padrão em relação à média e maior variação.

Tabela 17 – Resultado estatístico da leitura escalar em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.0007, 0.041, 0.163	0.068	0.084	1.01
2Mb	0.0007, 0.081, 0.455	0.18	0.24	1.10
4Mb	0.0008, 0.166, 0.9	0.35	0.48	1.09
8Mb	0.0008, 0.33, 1.91	0.75	1.02	1.11
16Mb	0.0008, 0.63, 4.08	1.57	2.19	1.14
32Mb	0.0008, 1.27, 7.87	3.04	4.22	1.13
64Mb	0.0008, 2.53, 15.83	6.12	8.50	1.13

6.3.2.2 Leitura de dados escalares em disco

Os testes de leitura seguintes foram para escalares em disco. O Gráfico 59 exibe os resultados encontrados.

Figura 58 – Desempenho de leitura de dados escalares em memória

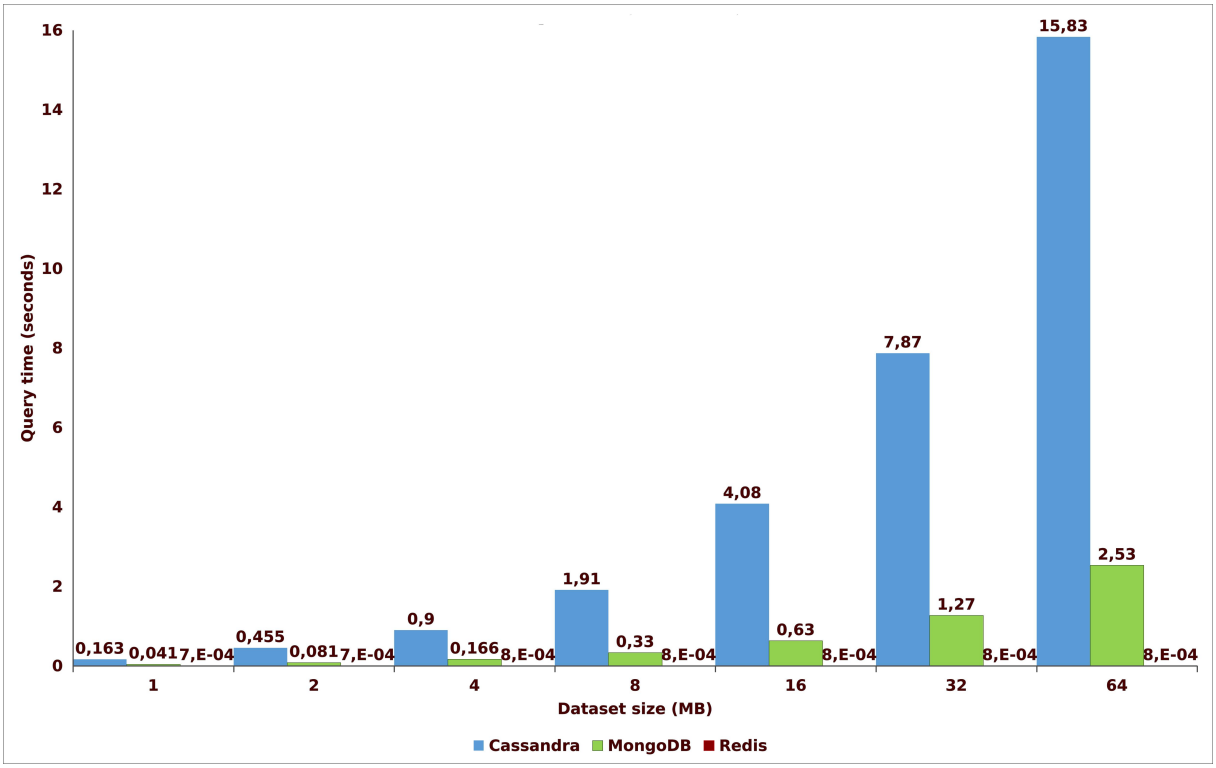
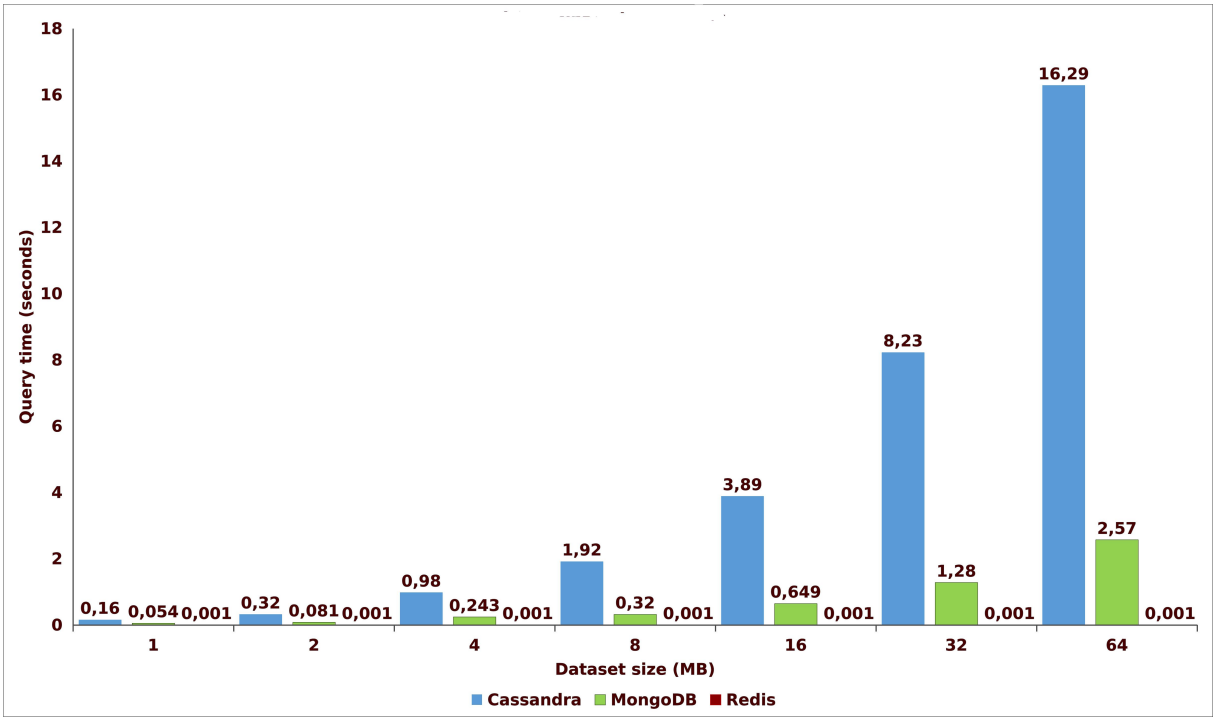


Figura 59 – Desempenho de leitura de dados escalares em disco



Após as 10 leituras de cada conjunto de dados, o Redis obteve menor tempo de busca que o MongoDB e Cassandra em todos os conjuntos de dados para também para leitura em disco.

O Redis obteve em média 74142% de desempenho superior ao banco de dados MongoDB e em média 454042% de desempenho superior ao banco de dados Cassandra. A Tabela 18 indica os resultados estatísticos encontrados. O grupo de 32Mb apresentou maior dispersão dos dados por possuir maior coeficiente de variação.

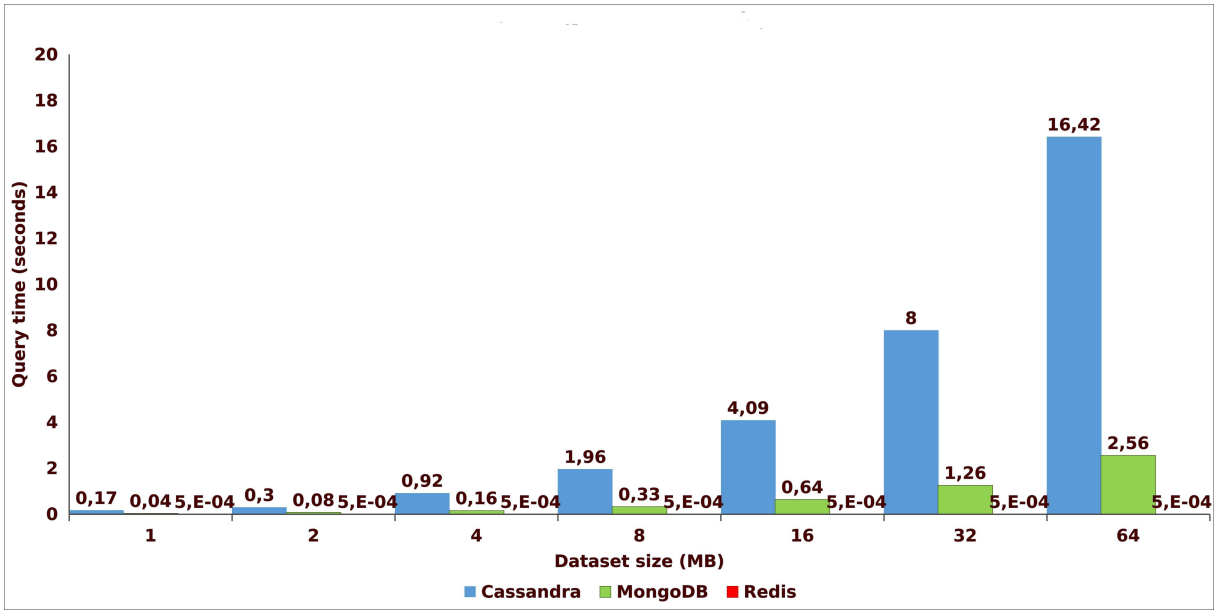
Tabela 18 – Resultado estatístico da leitura escalar em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.001, 0.054, 0.16	0.071	0.08	0.92
2Mb	0.001, 0.081, 0.32	0.13	0.16	1.01
4Mb	0.001, 0.243, 0.98	0.40	0.50	1.02
8Mb	0.001, 0.32, 1.92	0.74	1.02	1.12
16Mb	0.001, 0.649, 3.89	1.51	2.08	1.12
32Mb	0.001, 1.28, 8.23	3.17	4.42	1.14
64Mb	0.001, 2.57, 16.29	6.28	8.75	1.13

6.3.2.3 Leitura de dados posicionais em memória

Os testes de leitura seguites foram em dados posicionais nos bancos em memória. O Gráfico 60 exibe os resultados encontrados.

Figura 60 – Desempenho de leitura de dados posicionais em memória



Após as 10 leituras de cada conjunto de dados com cada banco de dados, mais uma vez, o banco de dados Redis superou os demais. O Redis obteve em média 144757% de desempenho superior ao MongoDB e em média 910185% de desempenho superior ao Cassandra. O grupo de 64Mb apresentou maior dispersão dos dados por possuir maior coeficiente de variação.

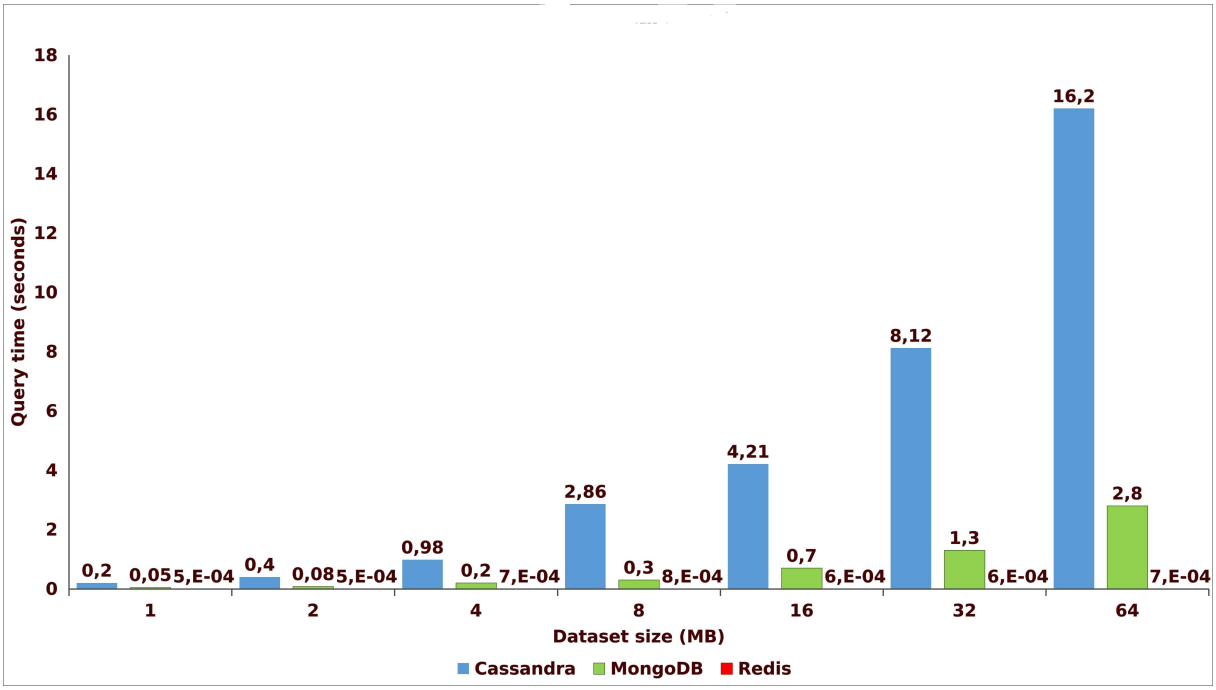
Tabela 19 – Resultado estatístico da leitura posicional em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.0005, 0.04, 0.17	0.070	0.088	1.03
2Mb	0.0005, 0.08, 0.3	0.12	0.15	0.99
4Mb	0.0005, 0.16, 0.92	0.36	0.49	1.11
8Mb	0.0005, 0.33, 1.96	0.76	1.04	1.12
16Mb	0.0005, 0.64, 4.09	1.57	2.19	1.13
32Mb	0.0005, 1.26, 8	3.08	4.30	1.13
64Mb	0.0005, 2.56, 16.42	6.32	8.83	1.14

6.3.2.4 Leitura de dados posicionais em disco

Os testes de leitura seguintes foram para posicionais em disco. O Gráfico 61 exibe os resultados encontrados.

Figura 61 – Desempenho de leitura de dados posicionais em disco



Após as 10 leituras de cada conjunto de dados com cada banco de dados, o banco de dados Redis superou os demais também na leitura posicional em disco. O Redis obteve em média 117814% de desempenho superior ao MongoDB e em média 712297% de desempenho superior ao Cassandra. A Tabela 20 indica os resultados estatísticos encontrados. O grupo de 8Mb apresentou maior dispersão dos dados por possuir maior coeficiente de variação.

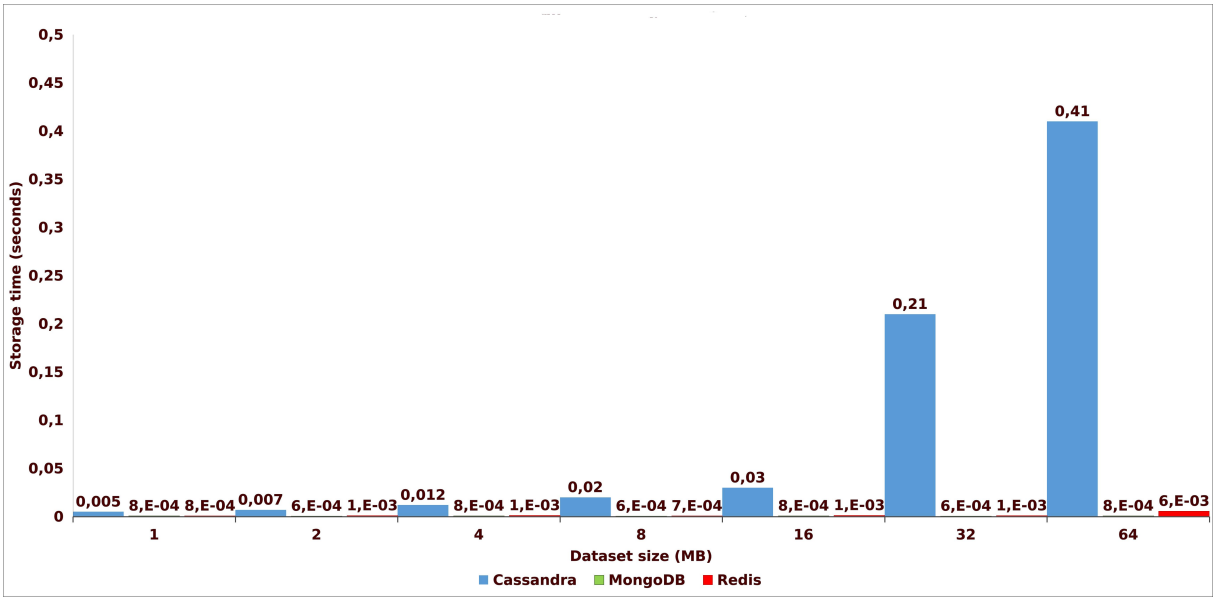
6.3.2.5 Leitura de dados multimídia em memória

Os testes seguintes foram de leitura de dados multimídia nos bancos configurados para armazenar em memória. O Gráfico 62 exibe os resultados encontrados.

Tabela 20 – Resultado estatístico da leitura posicional em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.0005, 0.05, 0.2	0.083	0.10	1.01
2Mb	0.0005, 0.08, 0.4	0.16	0.21	1.07
4Mb	0.0007, 0.2, 0.98	0.39	0.51	1.07
8Mb	0.0008, 0.3, 2.86	1.05	1.57	1.21
16Mb	0.0006, 0.7, 4.21	1.63	2.25	1.12
32Mb	0.0006, 1.3, 8.12	3.14	4.36	1.13
64Mb	0.0007, 2.8, 16.20	6.33	8.65	1.11

Figura 62 – Desempenho de leitura de dados multimídia em memória



Após as 10 leituras de cada conjunto de dados com cada banco de dados, diferente de todos os cenários anteriores, o MongoDB obteve menor tempo de busca que o Redis e Cassandra em todos os conjuntos de dados. O MongoDB obteve em média 127% de desempenho superior ao banco de dados Redis e em média 13703% de desempenho superior ao banco de dados Cassandra. A Tabela 21 indica os resultados estatísticos encontrados. O grupo de 32Mb apresentou maior dispersão dos dados por possuir maior coeficiente de variação.

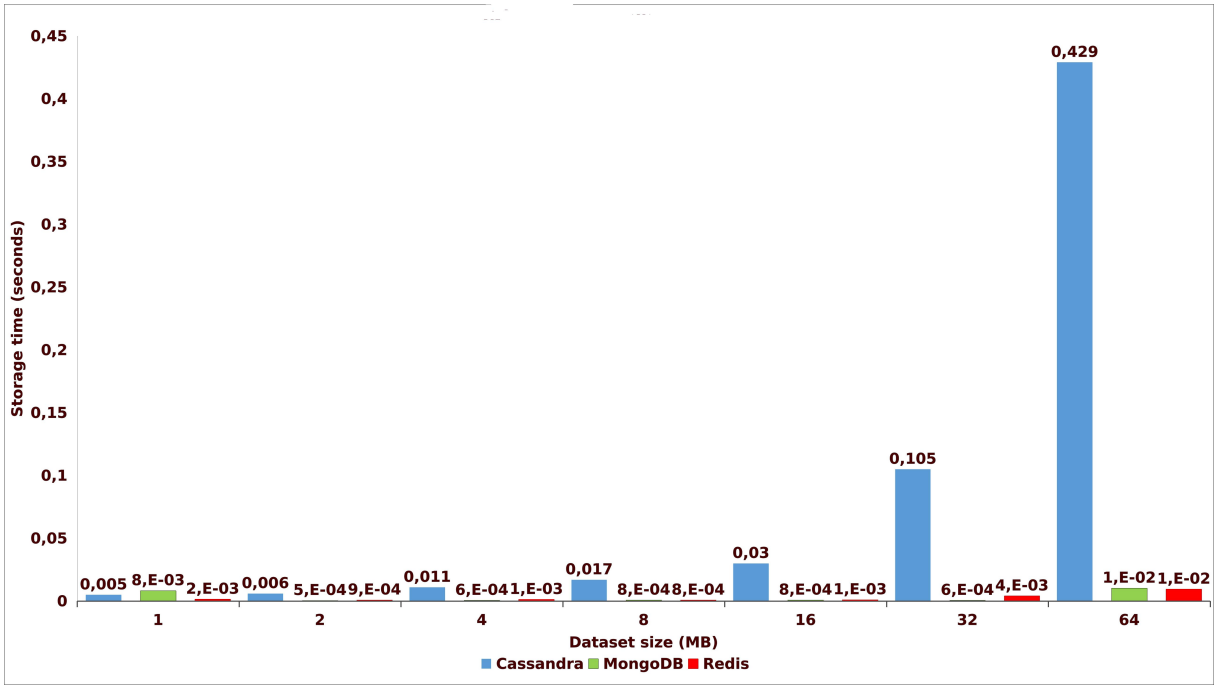
Tabela 21 – Resultado estatístico da leitura multimídia em memória

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.0008, 0.0008, 0.005	0.0022	0.0024	0.89
2Mb	0.0011, 0.0006, 0.007	0.0029	0.0035	1.00
4Mb	0.0012, 0.0008, 0.012	0.0046	0.0063	1.11
8Mb	0.0007, 0.0006, 0.02	0.0071	0.011	1.28
16Mb	0.0013, 0.0008, 0.03	0.010	0.016	1.27
32Mb	0.0010, 0.0006, 0.21	0.070	0.12	1.39
64Mb	0.0057, 0.0008, 0.41	0.13	0.23	1.38

6.3.2.6 Leitura de dados multimídia em disco

Os testes seguintes foram de leitura de dados multimídia nos bancos configurados para armazenar em disco. O Gráfico 63 exibe os resultados encontrados.

Figura 63 – Desempenho de leitura de dados multimídia em disco



Após as 10 leituras de cada conjunto de dados com cada banco de dados, o MongoDB obteve menor tempo de busca que o Redis e Cassandra no geral. O MongoDB obteve em média 103% de desempenho superior ao banco de dados Redis e em média 4276% de desempenho superior ao banco de dados Cassandra A Tabela 22 indica os resultados estatísticos encontrados. Os grupos de 32Mb e 64Mb apresentaram maior dispersão dos dados por possuirem maiores coeficientes de variação.

Tabela 22 – Resultado estatístico da leitura multimídia em disco

Grupo	Dados do Grupo	Média ($\bar{x}$)	Desvio Padrão (s)	Variação (c.v.%)
1Mb	0.0015, 0.0083, 0.005	0.0049	0.0034	0.56
2Mb	0.0009, 0.0005, 0.006	0.0024	0.00430	1.01
4Mb	0.0013, 0.0006, 0.011	0.0043	0.0058	1.10
8Mb	0.0008, 0.0008, 0.017	0.0062	0.009	1.23
16Mb	0.0011, 0.0008, 0.030	0.010	0.016	1.28
32Mb	0.0041, 0.0006, 0.105	0.036	0.059	1.32
64Mb	0.0095, 0.0103, 0.429	0.14	0.24	1.32

A Tabela 23 lista os resultados gerais dos experimentos. É possível notar que o Redis obteve superioridade em praticamente todos os cenários do experimento. Os únicos casos onde houve menor desempenho do Redis foram nas leituras de dados multimídia em memória e em disco, onde o MongoDB obteve o maior desempenho.

Tabela 23 – Resultados dos Testes

Resultados dos Testes
O Redis obteve melhor desempenho na **escrita de dados escalares em memória**.
O Redis obteve melhor desempenho na **escrita de dados escalares em disco**.
O Redis obteve melhor desempenho na **escrita de dados posicionais em memória**.
O Redis obteve melhor desempenho na **escrita de dados posicionais em disco**.
O Redis obteve melhor desempenho na **escrita de dados multimídia em memória**.
O Redis obteve melhor desempenho na **escrita de dados multimídia em disco**.
O Redis obteve melhor desempenho na **leitura de dados escalares em memória**.
O Redis obteve melhor desempenho na **leitura de dados escalares em disco**.
O Redis obteve melhor desempenho na **leitura de dados posicionais em memória**.
O Redis obteve melhor desempenho na **leitura de dados posicionais em disco**.
O MongoDB obteve melhor desempenho na **leitura de dados multimídia em memória**.
O MongoDB obteve melhor desempenho na **leitura de dados multimídia em disco**.

Por último, pode-se destacar que os resultados indicam que a arquitetura de armazenamento híbrido é uma abordagem promissora para enfrentar os desafios de desempenho e escalabilidade para armazenar dados de IoT. Diferentes mecanismos NoSQL foram utilizados para armazenar dados de IoT dos tipos escalares, posicionais e multimídia. Os resultados indicam que o banco de dados Redis, no geral, é o mais adequado para armazenar esses tipos de dados quando o desempenho é o principal critério utilizado.

6.3.3 Discussão

Para a realização dos experimentos, uma série de obstáculos ocorrerram, demandando diversas soluções de contorno, a fim de se conseguir cumprir o planejamento de execução do mesmo. Desde a troca de ambientes, até os ajustes em cada banco de dados, pode-se dizer que nenhum dos armazenamentos completou o experimento apenas com sua configuração padrão.

Inicialmente foi utilizado o Sistema Operacional Windows 10, onde cada banco de dados foi configurado para a realização dos testes. Nele, foram instalados e devidamente configurados os bancos de dados Redis, MongoDB, Cassandra e Neo4j. Na época, não se pensava em diferenciar o tipo de armazenamento em memória e em disco, com isso, apenas o banco Redis gravava em memória nativamente. Além dos bancos de dados, nessa época estava em desenvolvimento inicial a API em Python responsável por gerar a massa de dados para o testes e também por efetivamente executá-los.

Para a geração das massas de dados, considerou-se utilizar, como exemplos de dados escalares, valores de temperaturas, que variam de 0°C a 100°C. Para dados posicionais, latitudes como 26°60'N foram usadas, onde os esses valores numéricos que fazem parte do campo variam de 0 a 100. Após geradas as massas de dados com cada tamanho desejado, testes iniciais de escrita foram realizados. Já nesse momento dos testes, o banco de dados Redis teve uma aparente superioridade em relação ao demais. Essa superioridade causou a desconfiança de que o real motivo seria o Redis gravar em memória nativamente, enquanto os demais gravavam em disco, o que supostamente causaria uma perda de desempenho dos demais. Diante dessa situação surgiu

a necessidade de se realizar os testes com configurações separadas para gravação em disco e em memória em cada um dos bancos de dados. Foi também nesse momento que foi identificado que o banco de dados Neo4j tinham um tempo de escrita de cerca de 6x pior que o mais lento dos bancos testados até o momento, que era o Cassandra. Com isso, como o objetivo buscado era identificar o tipo de melhor desempenho, e considerando que aplicações críticas demandam rápida escrita e leitura de dados, o Neo4j foi excluído dos demais experimentos.

Num segundo momento dos experimentos, buscou-se a configuração dos bancos de dados Cassandra e MongoDB em memória, objetivando igualdade de condições entre os bancos. Contudo, um obstáculo encontrado foi que a configuração do MongoDB gravando em memória funciona apenas na versão *MongoDB Enterprise Server*, a qual só está disponível para sistemas Unix. Como o ambiente inicial estava em Windows, foi necessário utilizar outro sistema operacional, baseado em Unix, para configurar novamente todos os bancos de dados e poder simular testes em memória e em disco. Não bastava apenas configurar e executar o MongoDB no Unix e comparar com os outros bancos no Windows, pois a mudança de ambiente pode ter influência no resultado dos testes. Ou seja, era essencial manter todos os bancos no mesmo ambiente e em igualdade de condições.

Para poder simular os cenários, inicialmente se configurou novamente todo o ambiente numa máquina virtual contendo o Sistema Operacional Ubuntu 16.04. A configuração dos bancos em memória e em disco funcionou bem na máquina virtual, contudo, a mesma possuia apenas 4Gb de memória, metade do total disponível na máquina hospedeira. Com isso, mais uma vez, desconfiou-se de que os experimentos executados em uma máquina virtual possuam desempenho diferente de quando executado em máquina local. Para eliminar essa desconfiança, todo o ambiente foi novamente configurado diretamente na máquina hospedeira, ou seja, sem máquina virtual, sendo a que a máquina hospedeira possuia o sistema Unix Lubuntu, na versão 16.04. Com isso, atendeu-se o requisito de configurar todos os bancos de dados num máquina Unix, não virtual, em igualdade de condições. Logo após ter a configuração final do ambiente no Sistema Operacional Lubuntu 16.04, os experimentos em disco e em memória foram realizados nos bancos Redis, MongoDB e Cassandra, para dados escalares e posicionais. É importante destacar que como o Redis grava nativamente em memória, foi necessário mudar sua configuração para gravar apenas em disco quando os demais bancos de dados estavam configurados para gravar também apenas em disco.

Num momento posterior, após testar dados escalares e posicionais em memória e em disco, faltavam os testes com dados multimídia. Na API Python, os dados escalares foram tratados como tipos numéricos, os posicionais como *strings* e os dados multimídia como dados binários. Com isso, foi necessário investigar como cada um dos bancos de dados trata o armazenamento de dados binários, e se seria necessária alguma configuração adicional para lidar com dados binários, o que de fato ocorreu, como será descrito a seguir.

Inicialmente, foram feitos testes de escrita de multimídia no Redis, utilizando arquivos

de vídeo no tamanho desejado, como citado na seção de inserção de dados. Não foi necessário realizar nenhuma configuração adicional no Redis para gravar os arquivos binários de vídeo. Num segundo momento, foi experimentada a escrita dos vídeos no MongoDB. Para o MongoDB, houve um obstáculo que precisou ser resolvido. O MongoDB só aceita dados em BSON até o tamanho de 16Mb. No testes com dados maiores que esses, como na escrita de arquivos de vídeo com 32Mb e 64Mb, o MongoDB retorna erro, não permitindo a gravação. Para os casos de arquivos maiores que 16Mb, o MongoDB informa que deve ser utilizado os dados na formatação em *GridFS* (MONGODB, 2018a). Com isso, foi necessário ajustar a API em Python para trabalhar com *GridFS* e poder realizar todas as gravações dos arquivos de vídeo no MongoDB. Após os ajustes, os testes funcionaram como desejado.

Por fim, para finalizar os testes de escrita, estava faltando testar as inserções dos vídeos no Cassandra. O Cassandra, além de ser o banco com menor desempenho se comparado aos demais, foi o que mais teve problemas para realizar os testes com sua configuração padrão. Com isso, diversos parâmetros de configuração precisaram ser ajustados para permitir a inserção dos vídeos. O arquivo de configuração *cassandra.yaml* precisou ser ajustado, principalmente em relação ao aumento dos valores nos parâmetros referentes a *timeouts*. Além disso, também foi necessário aumentar o valor do parâmetro *commitlog_segment_size_in_mb* no arquivo *cassandra.yaml* para permitir gravar arquivos de vídeo maiores de 32Mb sem dar erros na gravação. Após os ajustes citados, os testes de escrita puderam ser concluídos também no Cassandra.

Para os testes de leitura, o foco foi buscar uma temperatura (dado escalar), uma latitude (dado posicional) ou um vídeo (da multimídia) em meio a um conjunto dados de 1Mb, 2Mb, 4Mb, 8Mb, 16Mb, 32Mb ou 64Mb. Teve-se o cuidado de garantir que o dado buscado tivesse apenas uma ocorrência no conjunto. Igualmente aos testes de escrita no Cassandra, na leitura de dados também houveram problemas em relação a expiração do tempo de operação. Com isso, também foi necessário aumentar os valores dos campos de *timeout* referente a consulta no arquivo *cassandra.yaml*. Para os demais bancos de dados, as consultas ocorreram sem problemas.

7 Conclusões e Trabalhos Futuros

Nesta seção são descritas as contribuições científicas obtidas, as linhas de pesquisa que podem ser alvo de trabalhos futuros além das publicações realizadas e em andamento.

7.1 Contribuições

1. Mapeamento do estado da arte acerca do armazenamento em IoT. Após a revisão sistemática foram obtidos os seguintes indicativos acerca dos 32 trabalhos relacionados selecionados:

   a) 78% utilizam um único banco de dados NoSQL.

   b) 59% trabalham com apenas um único tipo de dado de IoT entre os trabalhados na dissertação.

   c) 78% não descrevem dispositivos para coletar dados (sensores, *protoboards*, etc.).

   d) 25% definem uma arquitetura e realizam experimentos para validá-la.

2. Definição de uma nova arquitetura de armazenamento híbrido para IoT utilizando os principais tipos NoSQL existentes e os principais tipos de dados em IoT.

3. Validação do modelo proposto através de experimentos com foco em avaliar o desempenho de escrita e leitura de dados de IoT utilizando uma API proposta para realizar essa tarefa.

4. Realização da validação da arquitetura utilizando os principais tipos NoSQL existentes e os principais tipos de dados em IoT.

5. Após os experimentos, os resultados apontaram que o Redis superou os bancos MongoDB e Cassandra em praticamente todos os cenários, o que significa que, à excessão da leitura de dados multimídia em memória e em disco, este se mostra como a melhor escolha para esses tipos de dados quando o critério principal a ser adotado é o desempenho na escrita ou leitura de dados.

7.2 Trabalhos Futuros

1. Operacionalização da arquitetura com implementação dos principais métodos previstos.

2. Extensão do modelo para lidar com outros bancos de dados NoSQL.

3. Extensão do modelo para lidar com outros tipos de dados da IoT.

4. Realização dos experimentos em um servidor configurado na nuvem (Ex.: Amazon). Contrastar resultados do servidor remoto com os obtidos localmente.

5. Desenvolvimento de *drivers* para a camada de sensores para permitir a integração de sensores e *protoboards* (Ex.: Arduino, Raspberry Pi) à arquitetura.

6. Aplicação do modelo em estudo de caso real contendo algum dos cenários cobertos pela arquitetura, como o sensoriamento escalar (temperaturas, pressão, etc.), posicional ou modelos com armazenamento multimídia.

7.3 Publicações

Os resultados do trabalho foram publicados na conferência de Qualis A2 "ISCC - International Symposium on Computers and Communications" (IEEE, 2018). Adicionalmente, com a continuação da pesquisa e consequente amadurecimento do modelo e desenvolvimento das suas principais funcionalidades, espera-se publicar novos resultados no periódico de Qualis A2 "Future Generation Computer Systems" (ELSEVIR, 2018), periódico de grande expressão na área pesquisada, buscando dar maior visibilidade à pesquisa e facitar o acesso da mesma para novos pesquisadores.

Referências

AAZAM, M. et al. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In: IEEE. *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*. [S.l.], 2014. p. 414–419. Citado 2 vezes nas páginas 11 e 24.

AGENDA, I. *Find the IoT database that best fits your enterprise's needs*. 2016. Acessado em julho de 2018. Disponível em: <https://internetofthingsagenda.techtarget.com/feature/Find-the-IoT-database-that-best-fits-your-enterprises-needs>. Citado na página 12.

AMAZON. *Amazon Elastic Computing Cloud*. 2018. Acessado em julho de 2018. Disponível em: <aws.amazon.com/ec2>. Citado na página 21.

AMAZON. *Amazon Simpledb*. 2018. Acessado em julho de 2018. Disponível em: <https://aws.amazon.com/pt/simpledb/>. Citado na página 28.

APACHE. *Apache Cassandra*. 2018. Acessado em julho de 2018. Disponível em: <http://cassandra.apache.org/>. Citado 2 vezes nas páginas 28 e 35.

APACHE. *Apache Giraph*. 2018. Acessado em julho de 2018. Disponível em: <http://giraph.apache.org/>. Citado na página 37.

APACHE. *Apache HBase*. 2018. Acessado em julho de 2018. Disponível em: <http://hbase.apache.org>. Citado 2 vezes nas páginas 28 e 36.

APACHE. *Apache Spark*. 2018. Acessado em julho de 2018. Disponível em: <https://spark.apache.org/>. Citado 2 vezes nas páginas 37 e 70.

APACHE. *Apache TinkerPop*. 2018. Acessado em julho de 2018. Disponível em: <http://tinkerpop.apache.org/>. Citado na página 38.

APACHE. *CouchDB Relax*. 2018. Acessado em julho de 2018. Disponível em: <http://couchdb.apache.org>. Citado 2 vezes nas páginas 28 e 33.

ASHTON, K. That internet of things thing. *RFiD Journal*, v. 22, n. 7, p. 97–114, 2009. Citado na página 17.

ASLETT, M. How will the database incumbents respond to nosql and newsql. *San Francisco, The*, v. 451, p. 1–5, 2011. Citado 2 vezes nas páginas 12 e 68.

AYDIN, G.; HALLAC, I. R.; KARAKUS, B. Architecture and implementation of a scalable sensor data storage and analysis system using cloud computing and big data technologies. *Journal of Sensors*, Computer Engineering Department, Firat University, Elazig, Turkey, v. 2015, 2015. Citado 2 vezes nas páginas 54 e 66.

BESSANI, A. N. et al. Scfs: A shared cloud-backed file system. In: *USENIX ATC*. [S.l.: s.n.], 2014. p. 169–180. Citado 2 vezes nas páginas 12 e 68.

BOTTA, A. et al. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, Elsevier, v. 56, p. 684–700, 2016. Citado na página 11.

CAI, H. et al. A multi-layer Internet of things database schema for online-to-offline systems. *International Journal of Distributed Sensor Networks*, School of Software, Shanghai Jiao Tong University, Shanghai, China, v. 12, n. 8, 2016. Citado 2 vezes nas páginas 54 e 56.

CASSANDRA. *Cassandra*. 2018. Acessado em julho de 2018. Disponível em: <https://www.tutorialspoint.com/cassandra>. Citado na página 35.

CATTELL, R. Scalable sql and nosql data stores. *Acm Sigmod Record*, ACM, v. 39, n. 4, p. 12–27, 2011. Citado 5 vezes nas páginas 27, 28, 31, 34 e 35.

CETINTEMEL, U. et al. S-store: a streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 7, n. 13, p. 1633–1636, 2014. Citado 2 vezes nas páginas 12 e 68.

CUNHA, M.; FUKS, H. *AmbLEDs collaborative healthcare for AAL systems*. 2015. 626–631 p. Citado 2 vezes nas páginas 57 e 66.

DEVICES, A. *ADT7320*. 2018. Acessado em julho de 2018. Disponível em: <http://www.analog.com/en/products/analog-to-digital-converters/integrated-special-purpose-converters/digital-temperature-sensors/adt7320.html#product-overview>. Citado na página 20.

DEVICES, A. *CN0172*. 2018. Acessado em julho de 2018. Disponível em: <http://www.analog.com/en/design-center/reference-designs/hardware-reference-design/circuits-from-the-lab/cn0172.html>. Citado na página 20.

DÍAZ, M.; MARTÍN, C.; RUBIO, B. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer Applications*, Elsevier, v. 67, p. 99–117, 2016. Citado na página 24.

ELSEVIR. *Future Generation Computer Systems*. 2018. Acessado em julho de 2018. Disponível em: <https://www.journals.elsevier.com/future-generation-computer-systems>. Citado na página 92.

FAZIO, M. et al. Big Data Storage in the Cloud for Smart Environment Monitoring. *Procedia Computer Science*, v. 52, p. 500–506, 2015. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050915008236>. Citado 3 vezes nas páginas 44, 46 e 66.

FAZIO, M.; PULIAFITO, A.; VILLARI, M. IoT4S: A new architecture to exploit sensing capabilities in smart cities. *International Journal of Web and Grid Services*, Faculty of Engineering, University of Messina, Messina, Italy, v. 10, n. 2-3, p. 114–138, 2014. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84898621642{&}partnerID=40{&}md5=2592a3891b38a175cf52aede04>. Citado 2 vezes nas páginas 54 e 66.

FLEXISCALE. *FlexiScale Cloud Comp and Hosting*. 2018. Acessado em julho de 2018. Disponível em: <www.flexiscale.com>. Citado na página 21.

FOWLER, A. *NoSQL For Dummies*. [S.l.]: John Wiley & Sons, 2015. Citado 4 vezes nas páginas 12, 25, 26 e 68.

FOX, A. et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, v. 28, n. 13, p. 2009, 2009. Citado na página 22.

FRANCESCO, M. D. et al. A framework for multimodal sensing in heterogeneous and multimedia wireless sensor networks. In: IEEE. *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*. [S.l.], 2011. p. 1–3. Citado na página 57.

FRANCESCO, M. D. et al. *A Storage Infrastructure for Heterogeneous and Multimedia Data in the Internet of Things*. 2012. 26–33 p. Citado 3 vezes nas páginas 14, 57 e 66.

GANTZ, J.; REINSEL, D. Extracting value from chaos. *IDC iview*, v. 1142, p. 1–12, 2011. Citado na página 11.

GOGRID. *Cloud Computing Today*. 2018. Acessado em julho de 2018. Disponível em: <https://cloud-computing-today.com/category/gogrid/>. Citado na página 21.

GOOGLE. *Google App Engine*. 2018. Acessado em julho de 2018. Disponível em: <http://code.google.com/appengine>. Citado na página 21.

GOOGLE. *Google Bigtable*. 2018. Acessado em julho de 2018. Disponível em: <https://cloud.google.com/bigtable/>. Citado na página 28.

GOOGLE. *Lounge*. 2018. Acessado em julho de 2018. Disponível em: <http://code.google.com/p/couchdb-lounge/>. Citado na página 33.

GOOGLE. *Terrastore*. 2018. Acessado em julho de 2018. Disponível em: <https://code.google.com/archive/p/terrastore/>. Citado na página 28.

GROLINGER, K. et al. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: advances, systems and applications*, Springer, v. 2, n. 1, p. 22, 2013. Citado 2 vezes nas páginas 12 e 25.

GUBBI, J. et al. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, Elsevier, v. 29, n. 7, p. 1645–1660, 2013. Citado 3 vezes nas páginas 11, 17 e 23.

HASHI, Y. et al. *Design and Implementation of Data Management Scheme to Enable Efficient Analysis of Sensing Data*. 2015. 319–324 p. Citado 4 vezes nas páginas 5, 58, 59 e 66.

HECHT, R.; JABLONSKI, S. Nosql evaluation. In: IEEE. *International conference on cloud and service computing*. [S.l.], 2011. p. 336–41. Citado 3 vezes nas páginas 27, 30 e 34.

HONG, H. J. et al. *Optimizing Cloud-Based Video Crowdsensing*. 2016. 299–313 p. Citado 4 vezes nas páginas 5, 58, 59 e 66.

HUO, Z. et al. *Cloud-based Data-intensive Framework towards fault diagnosis in large-scale petrochemical plants*. 2016. 1080–1085 p. Citado 4 vezes nas páginas 45, 46, 47 e 66.

HYPERTABLE. *Hypertable*. 2018. Acessado em julho de 2018. Disponível em: <http://www.hypertable.org/>. Citado na página 28.

IEEE. *IEEE Symposium on Computers and Communications*. 2018. Acessado em julho de 2018. Disponível em: <http://iscc2018.ieee-iscc.org/>. Citado na página 92.

ISOTALO, J. Basics of statistics. *Finland: University of Tampere*, 2001. Citado na página 75.

IT, S. *DBENGINES*. 2018. Acessado em julho de 2018. Disponível em: <http://db-engines.com/en/ranking>. Citado 2 vezes nas páginas 29 e 73.

JIANG, L. et al. An IoT-Oriented Data Storage Framework in Cloud Computing Platform. *Industrial Informatics, IEEE Transactions on*, v. 10, n. 2, p. 1443–1451, 2014. ISSN 15513203. Citado 2 vezes nas páginas 45 e 47.

KANG, Y. S. et al. MongoDB-Based Repository Design for IoT-Generated RFID/Sensor Big Data. *IEEE Sensors Journal*, v. 16, n. 2, p. 485–497, 2016. ISSN 1530437X. Citado 3 vezes nas páginas 5, 59 e 66.

KEBAILI, M. O. et al. Landsliding Early Warning Prototype Using MongoDB and Web of Things Technologies. *Procedia Computer Science*, v. 98, p. 578–583, 2016. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050916322359>. Citado 4 vezes nas páginas 13, 56, 63 e 66.

LABS, F. *Tokyo Cabinet: a modern implementation of DBM*. 2018. Acessado em julho de 2018. Disponível em: <http://fallabs.com/tokyocabinet/>. Citado na página 27.

LE, T. D. et al. EPC information services with No-SQL datastore for the Internet of Things. *2014 IEEE International Conference on RFID (IEEE RFID)*, p. 47–54, 2014. Citado 2 vezes nas páginas 63 e 66.

LEAVITT, N. Will nosql databases live up to their promise? *Computer*, IEEE, v. 43, n. 2, 2010. Citado na página 25.

LI, T. et al. A storage solution for massive iot data based on nosql. In: IEEE. *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*. [S.l.], 2012. p. 50–57. Citado 2 vezes nas páginas 43 e 66.

LI, Y. et al. *Building a cloud-based platform for personal health sensor data management*. 2014. 223–226 p. Citado 2 vezes nas páginas 47 e 66.

LIN, Z. et al. *The Video Monitoring System Based on Big Data Processing*. 2014. 865–868 p. Citado 2 vezes nas páginas 48 e 66.

LIU, T.; WU, X.; LIAN, Y. *Design and construction of sunshine kitchen system based on cloud computing and video data analysis*. 2015. 211–215 p. Citado 4 vezes nas páginas 4, 48, 49 e 66.

LU, T.; FANG, J.; LIU, C. A unified storage and query optimization framework for sensor data. In: *Proceedings - 2015 12th Web Information System and Application Conference, WISA 2015*. Cloud Computing Research Center, North China University of Technology., Beijing Key Laboratory on Integration and Analysis of Large-scale Stream Data., Beijing, China: [s.n.], 2016. p. 229–234. ISBN VO -. Citado 2 vezes nas páginas 59 e 66.

LUAN, S. et al. *A Four-Layer Flexible Spatial Data Framework towards IoT Application*. 2015. 339–346 p. Citado 3 vezes nas páginas 49, 50 e 66.

MA, M. et al. *User-driven cloud transportation system for smart driving*. 2012. 658–665 p. Citado 2 vezes nas páginas 50 e 66.

MA, T.; MOTTA, G.; LIU, K. *Delivering Real-Time Information Services on Public Transit: A Framework*. 2017. 1–15 p. Citado 2 vezes nas páginas 50 e 66.

MEMCACHED. *What is Memcached?* 2018. Acessado em julho de 2018. Disponível em: <https://memcached.org/>. Citado na página 31.

MIORANDI, D. et al. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, Elsevier, v. 10, n. 7, p. 1497–1516, 2012. Citado 2 vezes nas páginas 17 e 18.

MOAWAD, A. et al. *Beyond discrete modeling: A continuous and efficient model for IoT*. 2015. 90–99 p. Citado 3 vezes nas páginas 5, 60 e 66.

MONGODB. *GridFS*. 2018. Acessado em julho de 2018. Disponível em: <https://docs.mongodb.com/manual/core/gridfs/>. Citado na página 90.

MONGODB. *MongoDB*. 2018. Acessado em julho de 2018. Disponível em: <http://mongodb.org>. Citado 2 vezes nas páginas 28 e 31.

MORENO-VOZMEDIANO, R.; MONTERO, R. S.; LLORENTE, I. M. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. *Computer*, IEEE, v. 45, n. 12, p. 65–72, 2012. Citado 2 vezes nas páginas 22 e 23.

NEO4J. *Neo4j*. 2018. Acessado em julho de 2018. Disponível em: <https://neo4j.com/>. Citado 2 vezes nas páginas 29 e 36.

NIMBITS. *com.nimbits*. 2018. Acessado em julho de 2018. Disponível em: <https://bsautner.github.io/com.nimbits/>. Citado na página 24.

NIST. *Cloud Computing*. 2018. Acessado em julho de 2018. Disponível em: <https://www.nist.gov/programs-projects/cloud-computing>. Citado na página 21.

ONTOTEXT. *GraphDB*. 2018. Acessado em julho de 2018. Disponível em: <http://graphdb.ontotext.com/>. Citado na página 29.

OPENIOT. *OpenIoT*. 2018. Acessado em julho de 2018. Disponível em: <http://www.openiot.eu/>. Citado na página 24.

OPENNEBULA. *OpenNebula*. 2018. Acessado em julho de 2018. Disponível em: <https://opennebula.org/>. Citado na página 22.

PHAN, T. A. M.; NURMINEN, J. K.; FRANCESCO, M. D. Cloud databases for internet-of-things data. In: IEEE. *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE*. [S.l.], 2014. p. 117–124. Citado 7 vezes nas páginas 11, 12, 14, 25, 66, 68 e 74.

PHAN, T. A. M. et al. Cloud databases for internet-of-things data. In: *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*. Department of Computer Science and Engineering, Aalto University School of Science, Finland: [s.n.], 2014. p. 117–124. ISBN VO -. Citado na página 61.

POSTSCAPES. *IoT Sensors and Actuators*. 2018. Acessado em julho de 2018. Disponível em: <https://www.postscapes.com/trackers/video/the-internet-of-things-and-sensors-and-actuators/>. Citado 2 vezes nas páginas 4 e 19.

RACKSPACE. *Openstack*. 2018. Acessado em julho de 2018. Disponível em: <https://www.openstack.org/>. Citado na página 23.

RACKSPACE. *Web Hosting by Rackspace Hosting*. 2018. Acessado em julho de 2018. Disponível em: <http://www.rackspace.com>. Citado na página 21.

RADU, C. et al. *Integrated Cloud Framework for Farm Management*. 2016. 302–307 p. Citado 2 vezes nas páginas 60 e 66.

RCRWIRELESSNEWS. *Sensor types and their IoT use cases*. 2018. Acessado em julho de 2018. Disponível em: <http://www.rcrwireless.com/20161206/internet-of-things/sensor-iot-tag31-tag99>. Citado na página 19.

REDIS. *Redis*. 2018. Acessado em julho de 2018. Disponível em: <https://redis.io/>. Citado 2 vezes nas páginas 27 e 29.

ROMAN, R.; ZHOU, J.; LOPEZ, J. On the features and challenges of security and privacy in distributed internet of things. *Computer Networks*, Elsevier, v. 57, n. 10, p. 2266–2279, 2013. Citado na página 18.

RONKAINEN, J.; IIVARI, A. Designing a data management pipeline for pervasive sensor communication systems. In: *Procedia Computer Science*. [S.l.: s.n.], 2015. v. 56, n. 1, p. 183–188. Citado 2 vezes nas páginas 51 e 66.

SALESFORCE. *Salesforce CRM*. 2018. Acessado em julho de 2018. Disponível em: <http://www.salesforce.com/platform>. Citado na página 21.

SAP. *SAP Business ByDesign*. 2018. Acessado em julho de 2018. Disponível em: <www.sap.com/sme/solutions/businessmanagement/businessbydesign/index.epx>. Citado na página 21.

SON, Y. S. et al. *Crowd-sourcing home energy efficiency measurement system*. 2015. 1272–1275 p. Citado 2 vezes nas páginas 44 e 66.

SPARKFUN. *HTU21D*. 2018. Acessado em julho de 2018. Disponível em: <https://www.sparkfun.com/products/retired/12064>. Citado na página 20.

ST. *LP25H*. 2018. Acessado em julho de 2018. Disponível em: <www.st.com/resource/en/datasheet/DM00066332.pdf>. Citado na página 20.

STONEBRAKER, M. Sql databases v. nosql databases. *Communications of the ACM*, ACM, v. 53, n. 4, p. 10–11, 2010. Citado 3 vezes nas páginas 11, 12 e 68.

TECHNOLOGIES, B. *Riak*. 2018. Acessado em julho de 2018. Disponível em: <basho.com/products/riak-kv/>. Citado na página 27.

TITAN. *Titan*. 2018. Acessado em julho de 2018. Disponível em: <http://titan.thinkaurelius.com/>. Citado 2 vezes nas páginas 29 e 37.

TIWARI, S. *Professional NoSQL*. [S.l.]: John Wiley & Sons, 2011. Citado 6 vezes nas páginas 26, 30, 31, 33, 34 e 36.

TUTORIALSPOINT. *Hadoop Tutorial*. 2018. Acessado em julho de 2018. Disponível em: <https://www.tutorialspoint.com//hadoop/index.htm>. Citado 3 vezes nas páginas 36, 37 e 70.

TUTORIALSPOINT. *Neo4j Tutorial*. 2018. Acessado em julho de 2018. Disponível em: <https://www.tutorialspoint.com//neo4j/index.htm>. Citado na página 37.

TWITTER. *Introducing FlockDB*. 2018. Acessado em julho de 2018. Disponível em: <https://blog.twitter.com/2010/introducing-flockdb>. Citado na página 29.

Van Der Veen, J. S. et al. Sensor data storage performance: SQL or NoSQL, physical or virtual. In: *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*. TNO, Groningen, Netherlands: [s.n.], 2012. p. 431–438. ISBN 2159-6182 VO -. Citado na página 62.

VANELLI, B. et al. *Internet of Things Data Storage Infrastructure in the Cloud Using NoSQL Databases*. 2017. 737–743 p. Citado 5 vezes nas páginas 5, 13, 56, 58 e 66.

VEEN, J. S. Van der; WAAIJ, B. Van der; MEIJER, R. J. Sensor data storage performance: Sql or nosql, physical or virtual. In: IEEE. *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. [S.l.], 2012. p. 431–438. Citado na página 66.

VICKNAIR, C. et al. A comparison of a graph database and a relational database: a data provenance perspective. In: ACM. *Proceedings of the 48th annual Southeast regional conference*. [S.l.], 2010. p. 42. Citado na página 29.

VOLDEMORT. *Project Voldemort*. 2018. Acessado em julho de 2018. Disponível em: <http://www.project-voldemort.com/voldemort/>. Citado na página 27.

WANG, Y. T. et al. *The Implementation of Sensor Data Access Cloud Service on HBase for Intelligent Indoor Environmental Monitoring*. 2016. 234–239 p. Citado 4 vezes nas páginas 5, 60, 61 e 66.

WINDOWS. *Windows Azure*. 2018. Acessado em julho de 2018. Disponível em: <www.microsoft.com/azure>. Citado na página 21.

WORLD, R. W. *IoT sensors*. 2018. Acessado em julho de 2018. Disponível em: <http://www.rfwireless-world.com/Terminology/IoT-sensors.html>. Citado na página 19.

WORTMANN, F.; FLÜCHTER, K. et al. Internet of things. *Business & Information Systems Engineering*, Springer Fachmedien Wiesbaden, v. 57, n. 3, p. 221–224, 2015. Citado na página 18.

YANG, W.; WANG, J.; ZHAO, Q. *Physical objects registration and management for Internet of Things*. 2016. 8335–8339 p. Citado 2 vezes nas páginas 51 e 66.

ZDRAVESKI, V. et al. *ISO-Standardized Smart City Platform Architecture and Dashboard*. 2017. 35–43 p. Citado 2 vezes nas páginas 54 e 66.

ZHAI, J. et al. Harnessing the flow of ecological data across networks, middleware, and applications. In: IEEE. *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*. [S.l.], 2016. p. 129–134. Citado 2 vezes nas páginas 61 e 66.

ZHANG, N. et al. *HBaseSpatial: A Scalable Spatial Data Storage Based on HBase*. 2014. 644–651 p. Citado 3 vezes nas páginas 14, 63 e 66.

ZHANG, Q. et al. *A Universal Storage Architecture for Big Data in Cloud Environment*. 2013. 476–480 p. Citado 5 vezes nas páginas 5, 14, 61, 62 e 66.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, Springer, v. 1, n. 1, p. 7–18, 2010. Citado 2 vezes nas páginas 4 e 21.

ZHAO, D. et al. Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: IEEE. *IEEE (Big Data)*. [S.l.], 2014. p. 61–70. Citado 2 vezes nas páginas 12 e 68.

ZIB. *What is Scalaris?* 2018. Acessado em julho de 2018. Disponível em: <http://scalaris.zib.de/>. Citado na página 27.